

AD-A202 574



AN EDUCATIONAL EXPERT SYSTEM SHELL
INTEGRATING
OBJECT-ATTRIBUTE-VALUE TRIPLES AND FRAMES

THESIS

Eddy G. Clark
Captain, USAF

AFIT/GCE/ENG/88D-2

DTIC
ELECTE
JAN 17 1989
S
D

DISTRIBUTION STATEMENT

Approved for public release
Distribution Unlimited

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

89 1 17 123

AFIT/GCE/ENG/88D-2

①

DTIC
ELECTE
JAN 17 1989
S D
AD

AN EDUCATIONAL EXPERT SYSTEM SHELL
INTEGRATING
OBJECT-ATTRIBUTE-VALUE TRIPLES AND FRAMES

THESIS

Eddy G. Clark
Captain, USAF

AFIT/GCE/ENG/88D-2

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution	
Availability	
Dist	Spec
A-1	

Approved for public release; distribution unlimited

AN EDUCATIONAL EXPERT SYSTEM SHELL
INTEGRATING
OBJECT-ATTRIBUTE-VALUE TRIPLES AND FRAMES

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Engineering

Eddy G. Clark, B.S.
Captain, USAF

December 1988

Preface

The purpose of this thesis was to design and develop an educational expert system shell (ESS) based upon object-attribute-value (OAV) triples and frames. The developed ESS is to be used in part, or in whole, in the artificial intelligence (AI) sequence at AFIT. Currently, the introductory course in the AI sequence at AFIT uses Prolog. In this course, the basic concepts of expert system shells are examined using an ESS called BC3. BC3 was written in standard Clocksin and Mellish Prolog to allow AFIT students to examine its code and understand how the basic concepts of an ESS can be implemented. A more advanced ESS, which allows for frames and uncertainties, will expand the teaching advantages which are currently present in BC3.

This thesis was divided into four phases in order to make the project more manageable in the time period allowed. This also allowed progressive testing of each phase because each phase was more feature-enhanced than the previous phase.

The four phases were chosen to correspond logically to the requirements imposed upon this thesis. ABC is the result of follow-up research and development of the existing BC3 EES. It seemed only appropriate that the first phase should be to examine BC3 and make minor changes to improve its efficiency or readability without changing its functionality.

The next three phases of this thesis came from altering BC3 to a) allow for a better user-interface, b) allow for frame knowledge

representation, and c) allow for the representation of uncertain knowledge.

In designing and implementing ABC, and throughout the writing of this thesis, I owe a large amount of gratitude to my thesis advisor, Dr F.M. Brown. Dr Brown provided a lot of his time in discussing the pros and cons of different design considerations. He also pointed me to several sources of much needed information and provided me with a seemingly endless number of critiques of my written work. Dr Brown's support and motivation was a key factor in the success of this thesis. I am also indebted to LTC Charles Bisbee and Capt Dave Umphress for their inputs into this thesis effort. Their direction and support throughout this thesis, but especially in the early stages, was invaluable. I also owe an endless amount of thanks to my wife, Christine, for the patience, support, and motivation she provided me with throughout this thesis effort.

Eddy G. Clark

Table of Contents

	Page
Preface	ii
List of Figures	viii
Abstract	x
I. Introduction	1-1
Background	1-1
Problems With BC3	1-3
Assumptions	1-4
Scope	1-5
Methodology	1-6
Overview	1-8
II. Background	2-1
Definitions	2-1
Artificial Intelligence	2-1
Expert Systems	2-2
Knowledge Base	2-2
Production Rules	2-4
Frames	2-6
Object-Attribute-Value Triples	2-8
Uncertainties	2-8
Certainty Factors	2-9
Quantifiers	2-9
Probabilistic Confidence Factors	2-10
Inference Mechanisms and Methods	2-11
Forward Chaining	2-11
Backward Chaining	2-11
Other Elements of Inferencing	2-12
Prolog	2-13
Commercial Expert System Shells	2-13
KEE	2-14
M.1	2-18
Personal Consultant Plus	2-20
Goldworks	2-21

III. An Overview of ABC	3-1
Knowledge Representations	3-1
Facts	3-2
Rules	3-4
Askables	3-5
Frames	3-7
Working Memory	3-8
IV. BC3 - Phase One: The Origin of ABC	4-1
Knowledge Representations	4-1
The How Trace	4-3
The Why Trace	4-4
User Interface	4-5
BC3 Problems and Solutions	4-6
The is_known Predicate	4-7
Altering The Trace Mechanisms	4-9
Testing Solutions	4-9
V. Phase Two: ABC's User Interface	5-1
The Two Types of Users	5-1
The Command Line	5-2
The Developer's Interface	5-3
Single Quotes: An Alternate to Prolog Lists	5-4
ABC Commands For The Developer	5-5
The Mechanics Involved	5-6
The End-User's Interface	5-7
'get_reply' and 'readline' Predicates	5-8
Enumerated Askables	5-9
Summary	5-11

VI. Phase Three: Implementing Frames	6-1
The Frame Structure	6-1
The Frame-Base Language	6-2
Accessing a Frame Value	6-4
Adding a Value to a Slot	6-4
Deleting a Value From a Slot	6-5
Adding or Deleting Slots or Frames	6-6
Summary	6-6
VII. Phase Four: Uncertainties	7-1
Uncertainties in Rules and Facts	7-1
The Search Strategy	7-3
Altering the Trace	7-5
VIII. Testing ABC	8-1
Testing ABC Predicates	8-1
Testing ABC Using Prewritten Knowledge Bases	8-3
IX. Conclusions and Recommendations	9-1
Summary	9-1
Assessment	9-3
Recommendations	9-4
Quick Fixes	9-4
Long-Term Enhancements	9-5
Bibliography	BIB-1
Appendix A: The Original BC3.PRO Expert System Shell	A-1
Appendix B: ABC Source Code	B-1
Appendix C: ABC User's Manual	C-1
Appendix D: ABC Predicates	D-1
Appendix E: ABC Predicate Dependencies	E-1
Appendix F: PETS Knowledge Base	F-1

Appendix G: WINE Knowledge Base	G-1
Appendix H: Sample of M.1 Knowledge Base	H-1
Appendix I: Clocksin and Mellish Prolog Predicates	I-1
Vita	VITA-1

List of Figures

Figure		Page
2-1.	Expert System Components	2-3
2-2.	Example of a Meta Rule	2-6
2-3.	A Node in a Frame System	2-7
2-4.	General Frame Structure	2-8
2-5.	Example of Certainty Factor Scales	2-10
2-6.	How to Edit a Facet in KEE	2-14
2-7.	A Frame Written in KEE	2-16
2-8.	Production Rules Within Frames With KEE	2-17
2-9.	GoldWorks Architecture	2-23
2-10.	A Relational Symbolic Representation in Goldworks . . .	2-25
2-11.	User-Defined Assertions in Goldworks	2-25
3-1.	An Example of a Fact in ABC	3-2
3-2.	An Example of a Fact in ABC With Certain Factor	3-3
3-3.	Representations Center Around Rules	3-4
3-4.	Example of a Rule in ABC	3-5
3-5.	Example of Initial Askable in ABC	3-6
3-6.	An Example of an ABC Initial-Askable Prompt	3-6
3-7.	An Example of a Frame in ABC	3-7
4-1.	An Example of a Rule in BC3	4-2
4-2.	Prolog Equivalent to BC3 Rule	4-2
4-3.	An Example of an Askable in BC3	4-6
4-4.	BC3's is_known /3 Inferencing Steps	4-8
4-5.	BC3 Test Times	4-10
5-1.	Different Ways to Enter OAV Triples	5-5

5-2.	The Load Prompt in ABC	5-6
5.3.	The ABC get_reply Predicate	5-9
5.4.	The ABC readline Predicate	5-10
5.5	Prompt From BC3 Askable	5-10
5.6.	Prompt From ABC Askable	5-11
6.1.	ABC's Frame Structure	6-3
6.2.	A Typical ABC Frame	6-3
7.1.	Rules Concerning Calculating Certainties in ABC	7-2
7.2.	ABC's calculate_CF Predicate	7-3
7.3.	Proper Ordering of Rules in ABC	7-5
8.1.	Prolog Predicate Notation	8-2
9.1.	Recommended Structure of ABC Assertable	9-5

Abstract

This thesis project investigates the creation of an expert system shell which integrates object-attribute-value (OAV) triples with frames and implements the shell in standard Prolog. Additionally, the implemented expert system shell uses certainty factors, which allow it to perform inexact reasoning. The shell, named AFIT Backward Chainer, or ABC, represents its knowledge in facts, rules, and frames. ABC has an explanation facility that can explain how it derives a solution or why it asks particular questions when seeking information from the user.

The approach used in this thesis was to study and expand upon an educational expert system shell called BC3. BC3, a rule-based shell developed at AFIT, symbolizes its knowledge with OAV triples. Once the decision to expand upon BC3 was made, the thesis project was divided into four separate but interrelated phases. At the end of each phase, a working expert system shell was implemented, limited to the functions of the current and previous phases.

Phase one was to study the functions of BC3 in order to determine its strengths and weaknesses and how best to correct its weaknesses while fully utilizing its strengths. During this phase, the reasoning predicate was fine-tuned for better efficiency and the trace mechanisms for the "how" trace were modified to use the Prolog database instead of the Prolog stack.

Phase two investigated different approaches to creating a user interface for an expert system shell with the constraint of having to implement it in standard Prolog. In this phase, the requirement to enter

data in the form of a Prolog structure was eliminated. Additionally, questions with unique queries prompted the user to select a number from an enumerated list of valid answers. A command line scheme similar to Teknowledge's M.1 was also implemented. The use of the command line gives the user much more freedom and flexibility.

Phase three investigated current commercial expert system shells to determine how frames and uncertainties appeared to be operated from the perspective of the user. The introduction of frames, including the possibility of demon procedures, into the expert system shell was then completed.

Phase four involved investigating inexact reasoning and introducing this capability into ABC. The inference engine was adapted to allow certainty factors to be used in the knowledge base's facts and rules. This adaptation of the inference engine allowed multiple solutions with differing certainty factors for a single goal.

The behavior of ABC is similar to M.1 but lacks the latter's efficiency. The wine advisor written by Teknowledge to run on M.1 was adapted to run on ABC to show its capability.

An Educational Expert System Shell
Integrating
Object-Attribute-Value Triples and Frames

I. Introduction

This chapter introduces background on the need for an educational expert system shell to be developed at AFIT. It also points out some of the problem areas which exist in the educational expert system shell currently in use at AFIT, assumptions that influenced the development of the new expert system shell, and the scope of the development project. The chapter concludes by detailing the methodology that was used in the design and development of the AFIT Backward Chainer expert system shell and a general overview of the work which went into the thesis.

Background

Currently, the Electrical and Computer Engineering department at AFIT is teaching an introductory course in artificial intelligence (AI) in which the structure and use of expert system shells are studied. Traditionally, for most of the students in this course, this is the first exposure to AI, AI languages, and expert system shells. The AI language used in the course is the Clocksin and Mellish (5:111-137) dialect of Prolog (Programming in Logic). Prolog is nonprocedural; it is therefore difficult for students new to its structure to understand its capability without the aid of good examples.

Last year when this course was last taught, engineering students were shown an expert system shell written in Clocksin and Mellish Prolog and were tasked to use it in developing small expert systems. The shell, named BC3, was developed at AFIT for pedagogical purposes.

BC3 uses Object-Attribute-Value (OAV) triples to represent its knowledge in "IF A AND B, THEN C" production rules, where A, B, and C are OAV triples. OAV triples were first used on a large scale in the mid-1970s in a Stanford University project called MYCIN (4:8,23), an expert system to aid physicians in the diagnosis of blood diseases (4:13-18).

MYCIN has two important features which make it a useful reference source for this project. First, MYCIN uses OAV triples in lieu of other known knowledge representations. Secondly, since many of the facts and rules in MYCIN are based upon historic data and physicians' experiences, certainty factors are used to represent the relative confidence of these facts and rules (4:3-8).

One of the strong points of BC3 is its ability to explain to its user how a particular solution was reached via its "how" trace. In expert systems such as MYCIN, such information is indispensable. In addition to BC3's "how" trace facility, BC3 also allows its user to query the expert system shell for why it needs supplementary information whenever prompted for it.

Problems With BC3

Problems with the current version of BC3 include the following:

1. The user interface in BC3 requires the student who is developing the expert system to input OAV triples into his/her knowledge-base in the form of Prolog lists. This includes all triples entered as parts of facts, rules, or other structures of knowledge. The need for such lists detracts from the understandability, and therefore maintainability, of the knowledge base.

Additionally, when prompted with a query, the user interface in BC3 requires the end user to reply with an answer meeting Prolog's term-syntax. This restricts flexibility and ease of use.

2. The knowledge representation in BC3 is limited to facts and rules. This limitation restricts the expert systems which can be developed efficiently with this shell to those whose problem domain can be easily represented by this structure. In problem domains which can be represented by hierarchical organization with inheritance of values, knowledge structures known as frames are more natural and efficient (22:74-75, 11:123). The lack of frame-based knowledge representation in BC3 prevents its efficient use in such hierarchical problem-domains. Additionally, the lack of a frame-based knowledge representation in BC3 detracts from its primary function as a learning tool. Frames are a very common method of representing knowledge in today's expert systems and a frame-based language incorporated into BC3 would greatly enhance its effectiveness as a teaching aid.

3. The knowledge representation in BC3 does not account for facts or production rules having varying degrees of confidence or uncertainties. The lack of this ability to associate certainty factors to the facts and rules in the knowledge base restricts the use of BC3 to problem domains where a fact or rule is either true 100 percent of the time, or is not true at all. This is a serious limitation since there are many problem domains in which facts are not just true or false but have truth-values at various levels in-between. Again, as an instructional tool for graduate students, BC3 does not provide the utility necessary to illustrate this basic concept which is used in many of the expert system shells currently being employed by the AI community.

Assumptions

The underlying assumption used in designing and developing the ABC expert system shell was that it would be used primarily as an education tool. Like the BC3 shell, ABC is intended to be functional, well documented, with clarity as its primary objective; efficiency is a secondary objective.

Furthermore, it was assumed that ABC would not be an effort started from beginning, but an effort to enhance the existing BC3 in order to add the additional features mentioned later. It was further assumed that all the additional features were to be implemented using standard Clocksin and Mellish Prolog.

A literature search was performed to look at current commercial expert system shells to see how the knowledge engineer perceives the

interaction between rules, frames, and uncertain knowledge. It was assumed that those concepts used in these commercial shells which were beneficial and within the scope of this thesis were to be emulated.

Scope

The ABC expert system shell is a greatly enhanced and modified version of the current educational shell, BC3. ABC's function is to extend the knowledge representation currently found in BC3 to include frames and uncertainties and, in addition, to add a better user interface.

The user interface was designed and implemented to allow its users to express OAV triples in a format more natural than the Prolog list. It also made the answering of queries more natural by eliminating the need for any of the user's replies to be in Prolog's term-syntax format.

A user's manual and sample knowledge bases are included in the appendices to aid in the understanding, development, and maintenance of ABC knowledge bases.

Methodology

Since the design and development of ABC was accomplished using Prolog, a language excellently suited for rapid prototyping, the methodology this thesis pursued was a modified version of the rapid prototyping paradigm (19:150). Rapid prototyping is the development of executable code very quickly. Prolog supports this methodology because its structure, which will be covered in more detail in Chapter II,

allows predicates to be developed and tested quickly using top-down development techniques. Additionally, Prolog's inferencing mechanism also supports top-down programming. Prolog's inference engine "starts with the goal and applies a top down, left-to-right evaluation strategy" (21:15, 109-111, 121).

The traditional rapid prototyping paradigm has six steps (19:149-150). The six steps of the prototyping paradigm and how each of these steps will be approached are enumerated below.

1. Establish the Need - The first step is to determine if rapid prototyping is appropriate. For this thesis, this determination was based primarily upon the language being used.

2. Dividing the Task Into Subtasks - The project must be divided into logical divisions and a subset of requirements for one or more of these divisions generated. Traditional requirement analysis methods can usually be applied. During this step, ABC was broken down into four divisions corresponding to the three problem areas plus an initial area. These divisions were: i) improving the readability and efficiency of the expert system shell BC3 as it originally functioned, ii) adding a better user interface to BC3, iii) adding a frame structure and frame language to BC3 along with modifying BC3's inference mechanism to interact with frames, and finally, iv) modifying the code to add the ability to address uncertainties in rules and facts.

3. Design Specification - Step three consists of getting the subset of requirements created in step two converted to an "abbreviated" design specification. For this thesis effort, the design specification was not formally created. Instead, the user's manual for the expert

system shell along with two sample knowledge bases were created. The ABC expert system shell was tested in the final phase of testing using these two knowledge bases. This set of tests was considered sufficient to confirm that the program meets the requirements of the user's manual. Additionally, an index of ABC's predicates and their dependencies is provided as a supplement in the appendices.

4. Create, Test and Refine Software - Step four is the point at which software is actually written. The iterative steps "create, test and refine" (18:150) are applied. Others have defined the last part of step four as a cycle of its own called the "Run-Debug-Edit cycle" (16:96). During each iteration of this step, the results were thoroughly tested for logic and programming errors. This testing was carried out using the Prolog-1 and Arity Prolog interpreters.

5. Getting User Input - Step five consists of showing the prototype to the user for either approval or for suggestions on improvement. For this thesis effort, the thesis sponsor was assumed to be the user. During each iteration of this step, the expert system shell was demonstrated to the thesis sponsor for tentative acceptance for that particular step. During each phase of the project which corresponds to one iteration of this step, it was necessary to go back to previous phases and make changes.

6. Iteration of Steps 4 and 5 - Step six calls for an expansion on the requirements by adding another division, referred to in step two, and repeating steps four and five until all the requirements are covered. Since all the requirements were defined ahead of time in the user's manual (Appendix C), with an occasional minor modification when

necessary, this step required that steps four and five be reapplied to the next phase of the expert system shell until all the phases were completed.

Overview

The following chapters of this thesis will allow the reader to familiarize himself with basic definitions and concepts surrounding expert system shells in general, and the work which went into the ABC expert system shell specifically. Chapter II provides the background information which includes definitions. Chapter III provides a brief functional description of ABC and explains its basic knowledge representations. Chapters IV through VII give detailed information concerning design, coding, and then outlines the results of adding frames, certainty factors, and a better user interface to BC3. Chapter VIII deals with test results when ABC ran the sample knowledge bases in appendices F and G. Chapter IX sums up the thesis with conclusions and recommendations. The casual reader may wish to read the functional description of ABC (Chapter III) and then proceed on to the final chapter on conclusions and recommendations.

II. Background

This chapter summarizes of the current knowledge relevant to this thesis. The chapter starts with definitions of terms so that any ambiguities of terminology can be reduced if not eliminated. A brief discussion of the Prolog language follows. Finally, this chapter covers some of the current commercial expert system shells available today with special emphasis on how the user perceives the structure of the knowledge representation along with how uncertainties are handled.

Definitions

This section provides a brief explanation of the major technical terms used throughout this thesis. Expert systems, knowledge representation, and uncertainties are among the topics which are discussed.

Artificial Intelligence. There are many definitions of artificial intelligence (AI) in the literature today. There are those who believe that the ultimate goal of AI is to develop "intelligent" machines which can understand and reason like humans (2:3). However, this thesis will accept the fundamental goal of AI to be to develop computer programs "to think in a limited way and to perform reasoning operations that a human might ordinarily do" (8:6). Throughout this thesis, many methodologies and techniques concerning search, pattern matching, and knowledge representation will be covered. These methodologies and techniques are referred to as AI methodologies and AI techniques (16:5).

Expert Systems. Expert systems are computer programs which stores declarative and procedural knowledge in a manner that allows it to solve problems in a narrow domain by reasoning with its knowledge, querying its user, or acquiring data from sensors (16:53-54). The primary distinction between expert systems and algorithmic systems is that expert systems encapsulate "rules of thumb" or "heuristics." Expert Systems are usually employed trying to solve very "difficult and poorly understood" (22:17) problems; thus they make use of heuristics to simplify their task. Being able to use heuristics efficiently makes an expert system more practical than a conventional algorithmic approach. An example of a problem where heuristics simplifies the solving process would be the game of chess. The complexity of coding chess in a traditional algorithmic manner would make the execution of the program using today's computer resources unfeasible.

Generally, expert systems have three major components: a knowledge base, one or more inference engines, and a user interface (16:55-58). The way each of these components are usually structured is illustrated in Figure 2-1.

Knowledge Base. The knowledge base is where all the domain specific knowledge is located (1:19). This knowledge is usually both declarative and procedural. Declarative knowledge is known facts concerning some object or event. A conventional database uses declarative knowledge. Procedural knowledge, on the other hand, is what gives expert systems its power to reason. Procedural knowledge provides "information about

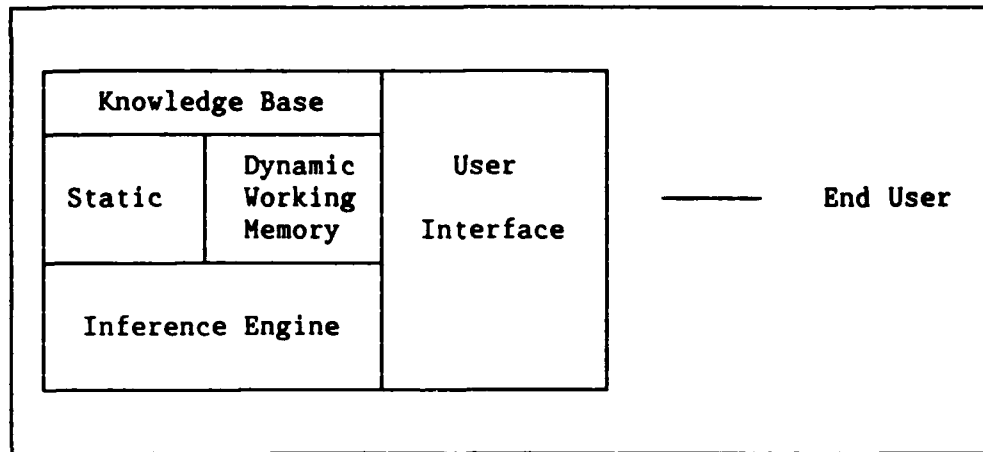


Figure 2-1: Expert System Components (16:55)

courses of action" (16:56). Rules of thumb or heuristics are usually a type of procedural knowledge (8:4). In the computer community, declarative knowledge is usually considered information. On the other hand, the useful combination of declarative knowledge and procedural knowledge is referred to as real knowledge. The primary object of expert systems can thus be summed up as trying to represent "knowledge rather than information in a computer" (8:5).

The structure of the knowledge base is often called its representation. The two most common knowledge base representations are production rules and frames; each will be explained later in this chapter. The manner in which information is represented in the knowledge base ultimately determines both the extent of domain knowledge which can be represented, and to a large degree, the difficulty of developing the knowledge base and the efficiency of it once it is developed. There are strong beliefs in the AI community concerning knowledge representations and the role these representations play in AI applications in the future. There are experts who believe "knowledge representation is the

central problem of AI" (1:19). One of the central themes "in the field of expert systems is largely devoted to finding better ways to represent knowledge in a computer" (8:6).

Some expert systems utilize more than one type of knowledge representation. For example, GoldWorks provides both production-rule and frame-based representations. Multiple knowledge representations provide greater flexibility and may extend the level of knowledge which may be encapsulated into the knowledge base (11:123).

Within each knowledge base representation -- facts, rules, or frames -- the knowledge is represented symbolically (8:9). Some convention of symbolic representation must be utilized in order that the AI techniques of search and pattern-matching, present in the inferencing mechanism, can make use of the knowledge residing in the knowledge base (8:9-10).

In some expert systems, the knowledge base is divided into two parts: the static knowledge, which remains essentially the same throughout each consultation, and the dynamic knowledge, consisting of knowledge learned during a consultation. The dynamic knowledge, more commonly called "working memory", can usually be stored to disk along with the static knowledge thus allowing future consultations the ability to take advantage of knowledge learned in past consultations (11:325).

Production Rules. Production rules are the most common type of knowledge representation used in expert systems. Production rules of the type, IF this AND that, THEN conclusion, have been around since 650 B.C. as a way to govern "everyday affairs" (4:12). They were first proposed in 1943 by Post as a general computational mechanism (4:20). They were

first used on a large scale expert system in the early 1970s. This expert system, called MYCIN, originally began as Edward H. Shortliffe's Ph.D. dissertation at Stanford University, which he finished in 1974 (4:xvii). MYCIN used the symbolic representation of Object-Attribute-Value (OAV) triples to represent its knowledge, with an additional argument to represent uncertainty (4:6-7). More will be said about OAV triples later. The domain knowledge was stripped from MYCIN and an expert system shell was formed called EMYCIN (4:xvii). M.1, an expert system shell from Teknowledge, was modeled around EMYCIN (11:303). Production rules are sometimes just called "rules" or "productions." The format of production rules has two parts: a premise and a conclusion. The premise and conclusion are sometimes referred to as the antecedent and consequent respectively. If the premise of a production can be proven true, then the conclusion is either accepted to be true or it causes some action to occur. It is the ease of representing heuristic knowledge in production rules which makes them the most popular representation in expert systems today (8:74-75).

Each production rule usually represents only a small portion of the knowledge needed in an expert system. For this reason, it is not unusual to have expert systems with hundreds or even thousands of rules (8:75).

Maintenance of production rules in expert systems is also an additional reason why rules are so popular. Besides being easy to create and modify, rules are relatively easy to maintain. This is increasingly important in large expert systems where the domain knowledge can change over short periods of time (8:75).

In some expert systems, there may be production rules which do not represent domain knowledge explicitly but instead provide knowledge about other rules. These types of production rules, usually called meta-rules, are rules which affect the control of the expert system (16:56). An example of a meta-rule, taken from MYCIN, is shown in Figure 2-2.

IF	1)	there are rules which do not mention the current goal in their premise
	2)	there are rules which mention the current goal in their premise
THEN		it is definite that the former should be done before the latter.

Figure 2-2: Example of a Meta Rule

Frames. Frames, like production rules, are a type of knowledge representation. The frame representation was first proposed by Marvin Minsky in 1975 as a way of representing and organizing "concepts and situations" (22:73). Frames can also be used for organizing control information. KEE, a very successful expert system shell, uses frames to organize control information and to organize its rules (12:1).

As a way to organize real-world concepts and situations, objects or ideas can be represented by a frame. Each frame can in turn have one or more different attributes, sometimes called slots, with each attribute having one or more values. Figure 2-3 shows the general structure of this type of frame (22:73-74).

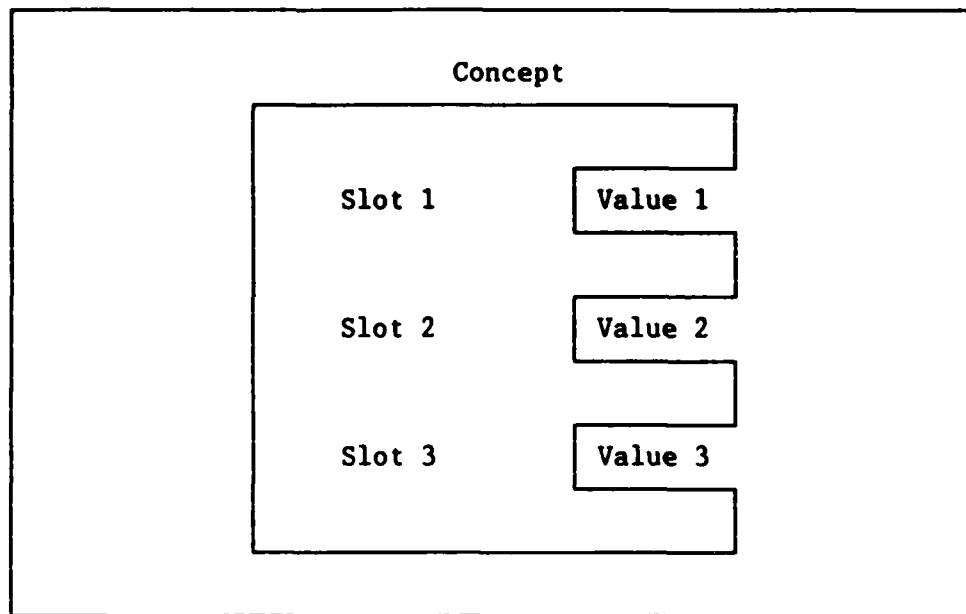


Figure 2-3: A Node in a Frame System (22:75)

One powerful feature of the frame representation is its ability to execute and run procedures, called demons, under some preset control strategy whenever a slot changes values or is called. Typical demons are "if-added", "if-removed", and "if-needed" (22:74-77). These demons are usually associated with slot facets such as value, default, if-needed, and if-added, which aid in the control strategy. Each slot can have one or more of these facets, allowing each slot to have one or more demons (8:82-85).

One of the more important features of a frame is its ability to be linked with other frames in some hierarchical fashion so that attributes and values are communicated naturally through a somewhat invisible control scheme. Usually, top level frames are general concepts or classes of objects and lower level frames are specific concepts or objects (22:73-74), as illustrated in the structure in Figure 2-4.

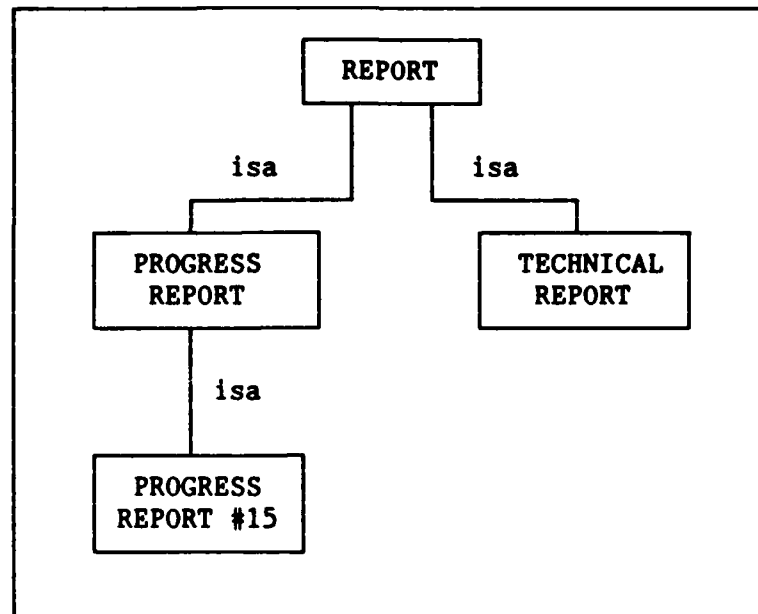


Figure 2-4: General Frame Structure (22:74)

Object-Attribute-Value Triples. An Object-Attribute-Value (OAV) triple is a structure designed to represent knowledge. This structure has three components: the object, the object's attribute, and the attribute's value. Thus information about an object, its attribute, and the value of the attribute can be arranged in a standardized structure enabling systematic and uniform processing.

Uncertainties.

There are generally three accepted methods for dealing with uncertainties in expert systems today: certainty factors, quantifiers, and probabilistic confidence factors (15:153). The use of uncertainties in knowledge representations allows for knowledge with various levels of confidence or truthfulness to be represented and manipulated in some

predesigned logical scheme. This capability is needed in many problem domains where the values of the knowledge lie somewhere between true and false. An example of a domain where uncertainties exist is medicine. MYCIN used certainty factors to deal with uncertainties in its medical diagnosis.

Certainty Factors. In most rule-based expert systems, the uncertainty of the knowledge is represented using certainty factors (8:79). The certainty factor (CF) is a numerical representation of the relative confidence that a fact or rule is indeed true or valid. In MYCIN, as with S.1, the certainty factors range from negative one to positive one. The negative-one CF represents absolute certainty that the knowledge is false. A positive-one CF, on the other hand, represents absolute certainty that the knowledge as presented is known to be true. A zero CF implies that nothing is known concerning the knowledge, i.e., that it is just as likely to be false as to be true (8:24,76-77).

The expert system shell M.1 uses a CF between zero and one-hundred. In this shell, zero represents that the knowledge is either unknown or that it is false. A CF of 100 represents absolute certainty that the knowledge is known to be true. M.1 interprets knowledge with a CF of less than 20 to be of questionable value and thus such knowledge is not used when it is needed as the condition of a rule (14:4-10). Both types of certainty factor scales are illustrated in Figure 2-5.

Quantifiers. English-like quantifiers, words or phrases, have a weighted representation. Common quantifiers are "some", "sure", "maybe", "most", and "always." Quantifiers allow expert systems with a natural

language front-end to parse the quantifier, equate it to a numerical representation internally, and when the answer is returned with its new numerical measure of confidence, it converts the new numerical measure back to a quantifier. Using quantifiers in expert systems is based upon "fuzzy logic" (15:153).

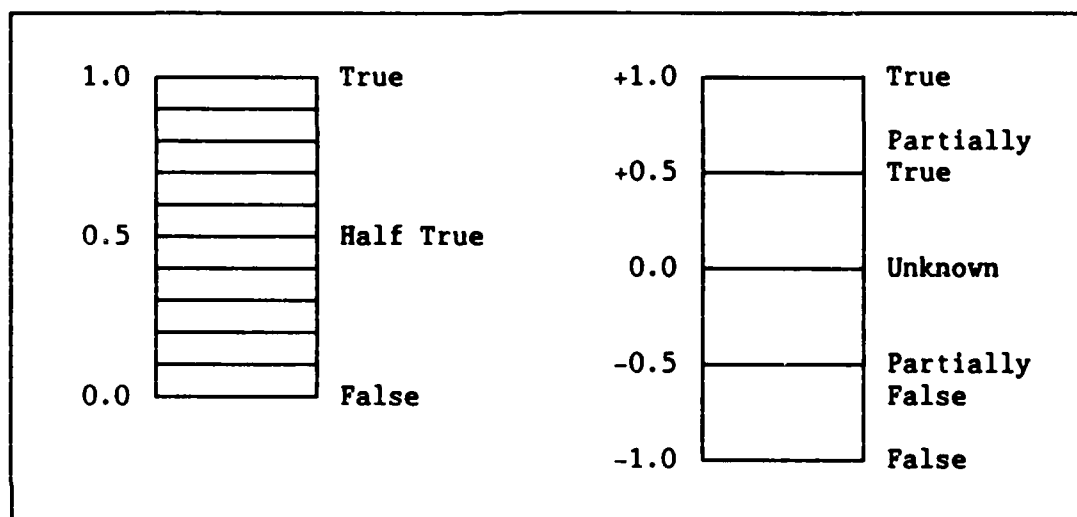


Figure 2-5: Example of Certainty Factor Scales (8:77)

Probabilistic Confidence Factors. The use of probabilistic confidence factors, based upon Bayesian probability, is the most math-intensive approach to dealing with uncertainty. A probabilistic confidence factor is a percentage of accuracy as opposed to a relative strength, which was the case with CF and fuzzy logic (15:153). The domain for the use of a Bayesian-type confidence representation is very narrow and several constraints must be met to get accurate results (20:245).

Inference Mechanisms and Methods

The inference engine is where "the process of creating explicit representations of knowledge from implicit ones" takes place (20:6). It is in the inference engine where heuristics, taken from the knowledge base, decide what path will be taken to find a solution. When external heuristics are not provided, the inferencing is very simple. Certainty factors are one type of external heuristics which can make the "decision-making" of the inference engine much more complex (11:245).

There are two general categories of inference engines: simple and complex. A simple inference engine is usually either forward chaining or backward chaining with a basic justification and search strategy. A complex inference engine usually uses both forward and backward inferencing or a complex justification or search technique (11:155-156). The basic elements in an inference engine, all of which are not required, are reviewed below.

Forward Chaining. Forward chaining inferencing is usually an iterative process of looking at each rule to see if its premise can match known facts. If so, the rule's conclusion is added to the list of known facts. This process continues until no rules have premises which can match known facts. Forward chaining can produce a large number of new facts; thus it is usually employed when a number of answers or solutions are needed from a fixed set of facts and rules (11:155-159).

Backward Chaining. In backward chaining inferencing, a goal is known but what the inferencing scheme tries to determine is if that goal can be deduced from the facts and rules within the knowledge base. This

process usually involves entering a goal into the knowledge base which matches one or more rule conclusions. If a rule conclusion is matched, the premises of that rule are made subgoals themselves. If all the premise subgoals are proven, this then proves that the original goal can be derived from the knowledge base. Throughout this process, variables become instantiated and a solution consisting of these instantiated variables is usually presented to the user.

If any of the premise subgoals fails, the inference engine goes to the next rule whose conclusion matches the queried goal. If no resolution can be made, then the original goal cannot be derived by the knowledge in the knowledge base (11:160-163). It is important to note that failure of resolution does not imply that the goal is false but merely that it can not be determined based on the facts and rules in the knowledge base.

Other Elements of Inferencing. Other common elements of inferencing are justification and search. There are three basic types of search mechanisms: depth-first, breadth-first, and best-first (11:155-159). Other search mechanisms can be formed from the more primitive ones.

Search mechanisms vary from simple depth-first or breadth-first searches to complex best-first type searches. Because of the "combinatorial explosion" of possible paths, some technique is required to reduce the number of paths only to those paths which look the most promising. This is done in games such as chess. This technique is sometimes referred to as pruning and can be accomplished using the more complex heuristic search mechanisms such as best-first (8:86-93).

Justification is a trace-type mechanism which records the track the inference engine takes as it reasons through the knowledge base (11:164). This is important in many expert systems where the question of how an answer is derived is as important as the answer itself (4:64).

Prolog

Prolog was originally developed under the guidance of a Frenchman named Alain Colmerauer around 1970. Prolog was the first language specifically designed to allow a programmer to express his programs in logic as opposed to more "conventional programming constructs about what the machine should do when" (5:233).

Prolog is a declarative language as opposed to a procedural language such as Fortran, Ada, or Pascal. In Prolog, the programmer is chiefly concerned with expressing his problem in logical expressions, (facts and rules), and not with the minute details of how things are accomplished (3:40-42). Thus, the declarative nature of Prolog makes it very suitable for expert systems and expert system shells (15:127).

Commercial Expert Systems Shells

This section provides a brief synopsis of a few of the commercial expert system shells, with particular attention to how these shells allow their users to write knowledge structures in the form of rules and/or frames, and how these structures deal with uncertain knowledge. The four expert system shells evaluated are KEE, M.1, Personal Consultant Plus, and GoldWorks.

KEE. KEE is an acronym for Knowledge Engineering Environment and is a trademark of Intellicorp Inc. KEE is an expert system "environment" written in Common Lisp. It requires the use of a Common Lisp interpreter in order to operate. The user is allowed to program using the KEE syntax which works inside of its environment and utilizes windows to display frame structures or relationships between frames in a graphical fashion. The KEE programmer may also input certain instructions using user-defined Common Lisp code which adds flexibility to the predefined KEE environment.

There are eleven types of windows the user can access. Each type can be accessed using the standard windowing interface or by a user-defined interface written in Common Lisp. For example, the editor window is used to "edit units, slots, slot values, facet values, rules, and ruleclasses" (12:1-38). Such a window is shown in Figure 2-6.

```
(edit-value-of-facet
  '(goat farmdil)      ; unit-or-ref
  'location            ; slot-or-ref
  'own                 ; slot-type
  'hide.me             ; facet
  'take-goat-to-shore1-9) ; world [12:1-39]
```

Figure 2-6: How to Edit a Facet in KEE

The KEE frame structure is a superset of the structure Dr. Minsky introduced. KEE not only uses frames for controlling the reasoning components of its environment, but uses frames to represent "structural representation issues" as well (7:907). The main components of KEE's

frame structure are class, inheritance, MemberOf link, slots, facets, values, and procedural attachments.

Each frame has one or more slots with each slot having one or more values. Classes, such as value classes, limit the value which any slot may acquire to preauthorized values. Closely related are the facets called CardinalityMin and CardinalityMax which are only two of several facets which KEE uses. These two facets limit the number of different values which a slot may acquire to a number between or including these two values.

Inheritance in KEE is made possible by the use of either MemberOf links or subclass links. Each frame can have one or more MemberOf links to other class frames and each class frame can have one or more subclass links to other class frames.

A procedural attachment "enables behavioral models of objects and expertise in an application domain to be built" (7:909). KEE has two types of procedural attachments: methods and active values. Messages can be sent to frames which have a "message-responder slot." The value of these message-responder slots become what KEE calls a method. Active values are very similar to demons. Active values fire off one or more rules if specific changes in the frame occur.

A typical frame written in KEE syntax is listed in Figure 2-7.

Each rule in KEE is in the form of a frame. The frame language is fully accessible to the programmer. Additionally, a predicate logic language is also available to supplement the predicates already available in Common Lisp. Predicates are used in the development of production rules. An example of a predicate in KEE's predicate logic language is

IN.CLASS, which has two arguments. The predicate is true if the first argument is a member of the class described by the second argument. An example of how production rules are written using KEE's predicate logic language is shown in Figure 2-8.

```
Unit: TRUCKS in knowledge base TRANSPORTATION
  Superclasses: VEHICLE
  Subclasses: BIG.NON.RED.TRUCKS, HUGE.GREY.TRUCKS
  Member: CLASSES.OF.PHYSICAL.OBJECTS

-----
.
.
.

MemberSlot: DIAGNOSE from TRUCKS
  Inheritance: METHOD
  ValueClass: METHODS
  Cardinality.Min: 1
  Cardinality.Max: 1
  Comment: "A method for diagnosing electrical faults"
  Values: TRUCK.DIAGNOSIS.FUNCTION

MemberSlot: ELECTRICAL.FAULTS from TRUCKS
  Comment: "Faults found by the DIAGNOSIS method"
  Values: Unknown

MemberSlot: LOCATION from PHYSICAL.OBJECTS
  Cardinality.Min: 1
  Cardinality.Max: 1
  Values: Unknown
  ActiveValues: UPDATE.LOCATION

.
.
.
```

Figure 2-7: A Frame Written in KEE (7:911)

Unit: BIG.NON.RED.TRUCKS.RULE in knowledge base TRANSPORTATION
Member: TRUCK.CLASSIFICATION.RULES

OwnSlot: ACTION from RULES
Inheritance: UNION
Values: Unknown

OwnSlot: ASSERTION from BIG.NON.RED.TRUCKS.RULE
Inheritance: UNION
ActiveValues: WFFINDEX
Values: |Wff:(?X IS IN CLASS BIG.NON.RED.TRUCKS)

OwnSlot: EXTERNAL.FORM from BIG.NON.RED.TRUCKS.RULE
Inheritance: SAME
ValueClass: LIST
ActiveValues: RULEPARSE
Values: (IF ((?X IS IN CLASS TRUCKS)
 AND
 (GREATERP (THE WEIGHT OF ?X)
 10000)
 AND
 (?X HAS AT LEAST 10 WHEELS)
 AND
 (NOT (THE COLOR OF ?X IS RED)))
 THEN
 (?X IS IN CLASS BIG.NON.RED.TRUCKS))

OwnSlot: PARSE from RULES
Inheritance: METHOD
ValueClass: METHODS
Values: DEFAULT.RULE.PARSER

OwnSlot: PREMISE from BIG.NON.RED.TRUCKS.RULE
Inheritance: UNION
ActiveValues: WFFINDEX
Values: |Wff:(?X IS IN CLASS TRUCKS)
 |Wff:(THE WEIGHT OF ?X IS ?VAR29)
 |Wff:(GREATERP ?VAR29 10000)
 |Wff:(?X HAS AT LEAST 10 WHEELS)
 |Wff:(NOT (THE COLOR OF ?X IS RED))

.
.
.

Figure 2-8: Production Rules Within Frames With KEE (7:913)

M.1. M.1 can be considered both an expert system shell and a logic programming language. The first version of M.1 was written in Prolog and released in 1984 and has sold over 4000 copies (10:1). Its creator, Teknowledge Inc., has released a second version which was rewritten in the C language.

M.1 uses production rules and does not have the capability of frame representation. It does, on the other hand, allow for uncertain knowledge by its support of certainty factors. M.1 has three "forms" of representation; facts, rules, and meta facts (14:1-12). Version One of M.1 used predefined Prolog structures to represent facts and rules. M.1 defined Prolog operators such as "default", "=", and "cf" which allowed facts to be written such as "the default color = red cf 50", where "color" is the attribute and "red" is its value with a certainty factor of 50 percent. BC3 uses Prolog to represent its facts and rules in much the same way. The largest difference between M.1 and BC3 is that M.1 used attribute-value pairs instead of object-attribute-value triples.

Meta facts in M.1 are facts "that M.1 uses to find another fact." Meta facts are facts which aid the inference engine in its search to find a solution to its goals (14:3-9). The 'askable' predicate in BC3 which allows the user to be queried for a confirmation or a value can be considered a meta fact of sorts.

The inference engine in M.1 is driven by three components: modus ponens, instantiation, and certainty factors (14:3-11). Modus ponens and instantiation are inferencing mechanisms derived directly from Prolog. Certainty factors (CF) in M.1 allow it to "reach conclusions using ... inexact information" (14:1-7).

Certainty factors in M.1 can be located in facts as well as rules. The certainty factor in an M.1 fact implies that the fact is known with a relative strength corresponding to the numerical weight of its certainty factor. M.1 uses the range from 0, signifying no certainty at all, to 100, signifying absolute certainty, to weight the certainty of its facts and rules (14:3-14).

Certainty factors in M.1 rules are more complex compared to their role in facts. When discussing certainty factors within rules, both the manner in which a particular rule obtains its final certainty factor and the manner in which certainty factors from two or more rules are combined are important.

The final certainty factor of a rule is normally the product of the certainty factor of the overall rule itself and the lowest certainty factor of the individual "if" clauses which make up the rule. Choosing the lowest certainty factor of the "if" clauses is analogous to rating the strength of a chain based upon its weakest link. This method of calculating the overall certainty factor of a rule applies only when the rule is exclusively made up of conjunctive "if" clauses (14:3-15, 3-16).

If the rule's clauses are disjunctive, "if A or B then C", then each disjunctive part of the rule is considered a separate rule. The final certainty factor for rules with disjunctive clauses cannot be obtained using the method described previously for rules having only conjunctive clauses. Multiple rules with the same conclusion, or rules with disjunctive clauses, reinforce each other. This reinforcement of two or more rules causes an assertion to be created in M.1's cache, temporary

working memory, with an overall certainty factor greater than the two or more certainty factors from which it was composed.

The method used in M.1 to calculate the final certainty factor of two or more rules with the same conclusion is to solve each of the rules independently and combine their certainty factors using equation (1) (14:3-17).

$$CF1 + (100 - CF1)/100 * CF2 \quad (1)$$

Two example M.1 knowledge bases are attached in Appendix B. These knowledge bases help show the structure of M.1 from the programmer's perspective. They also show how variables must be represented and how operators may be optionally defined (indicating that M.1 relies on the Prolog structure substantially).

Personal Consultant Plus. Personal Consultant Plus (PC Plus) is an expert system shell developed by Texas Instruments (TI) which is written in their implementation of Lisp, PC Scheme. In order for PC Plus to run, PC Scheme must be installed and loaded into the computer's memory (18:2-2).

PC Plus supports three types of knowledge structures: frames, parameters, and rules (18:3-1). The underlying structure in PC Plus is the frame. Rules and parameters are just two of the three major parts of a frame. The other major part of a frame is called a property. Frames are used to store the knowledge about a very narrow and specific problem domain. Each frame has its own facts, its own rules and its own properties. Parameters in PC Plus are basically facts. There are five

basic frame properties which the user may define: goals, promptever, translation, initialdata, and displayresults. Properties are used "to describe the features of a frame and to control various aspects of the consultation" (18:3-3,3-4).

Since the function of a frame in PC Plus is to store knowledge about a problem domain, the primary task for a frame is to come up with a solution to one or more goals. Thus, each frame has one or more parameters which make up the goal's property. Once these parameters are known, the frame's solution is known.

PC Plus frames use hierarchy to organize their knowledge. Each frame may be a parent frame, a child frame, or both. Parameters and rules from the parent frames are inherited by the child frames. The highest frame is called the root frame (18:3-5).

Uncertainties in PC Plus are also supported in the form of certainty factors (CF). CF may be in the range of -100 to 100 where -100 represents a parameter or a rule which is absolutely false. In contrast, a CF of 100 represents a parameter or a rule which is absolutely true. A CF of zero indicates that the parameter or rule neither supports or denies its assertion. When the certainty of a fact is totally unknown, it may be represented with a CF of zero (18:3-7). This CF scheme is similar to the one used with MYCIN.

GoldWorks. GoldWorks, an expert system shell from Gold Hill Computers Inc., is written in Gold Hill's Common Lisp. GoldWorks provides two main interfaces: the menu interface and the developer's interface.

The menu interface is a convenient way for the novice to interact with the many features which GoldWorks provides. The menu interface allows the developer or user to load and save files, enter a tutorial, enter the editor called GMACS, enter the DOS environment, define frames or rules, browse through the frame structure, and of course exit GoldWorks. It will not, however, provide all the flexibility which the developer's interface has.

The developer's interface allows access to the underlying Lisp interpreter. This allows the developer to write his own assertions, rules, or procedures in Lisp, adding both to the power and flexibility of the expert systems which can be developed using GoldWorks. Additionally, the developer's interface allows access to a screen toolkit allowing customized screens to be generated. For business applications, the developer's interface also allows access to Lotus 1-2-3 and dBase III+ files. The user interfaces are illustrated within the architecture of GoldWorks in Figure 2-9.

GoldWorks has three types of knowledge representations: Frames, Rules, and assertions. A frame must be created before instances of the frames can be generated; therefore, a frame is created with all the slots that are needed but with no values. There are nine types of facets which each slot may have: print name, documentation string, explanation string, constraint, multivalued, default, certainty, when-modified, and user. Each of these facets is defined.

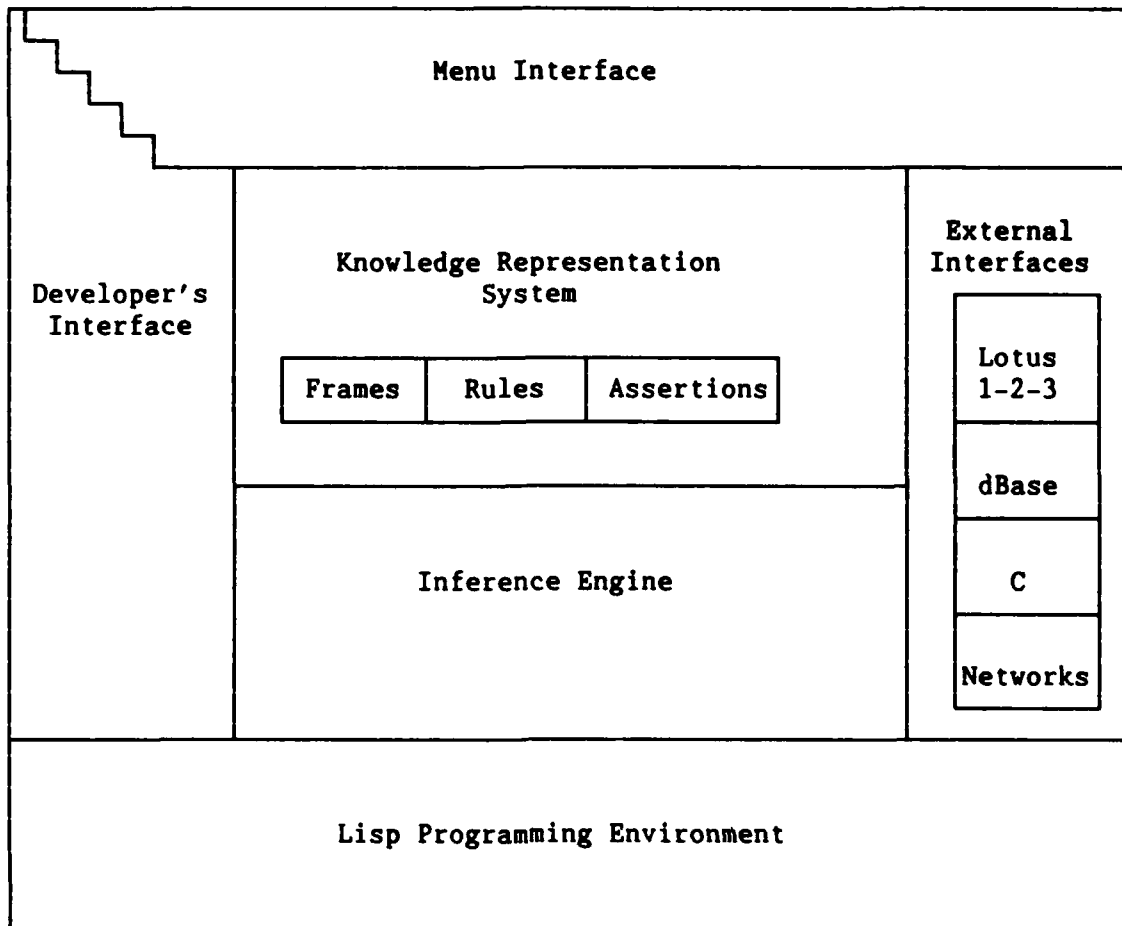


Figure 2-9: GoldWorks Architecture (13:71)

The print name and documentation string are administrative facets which describe the slot. The explanation string adds "text to the end of the system-generated explanation for slot-value assertions associated with the slot" (13:70). The constraint facet limits the value which a slot may acquire. The multivalued facet is either yes or no, indicating whether the slot has multiple values. The default facet simply provides a default value for instances of the frame which have no value for that particular slot. The certainty facet provides a default certainty factor to the slot. The when-modified facet is a default list of Lisp functions

which will be evaluated whenever the slot is modified. The user facet's function is defined by the developer.

When an instantiation of a frame is created, the slots along with facets of the uninstantiated frame are used. The instantiated frame will hold its values in its slots.

Rules in GoldWorks are used to infer implicit facts from explicit assertions. There are three main types of rules in GoldWorks: forward-chaining, backward-chaining, and bidirectional. Rules have eight facets: print name, document string, explanation string, direction, sponsor, priority, certainty factor, and dependency value. Each of these facets is defined below.

The print name, document string, explanation string, and certainty factor have the same purpose as the facets with the same name covered previously for frames. The direction facet simply signifies which of the three types of rules it is. The sponsor facet allows the grouping together of rules which may perform some function. The priority facet is an integer between -1000 and 1000 which prioritizes rules in the order they are to fire. The dependency value facet is either true or nil. If the dependency value is true, this

tells the system to create a justification (why the rule is in the knowledge base) when the rule fires. If assertions that match a rule (causing it to fire) are retracted, the assertions that resulted from firing that rule also will be retracted [13:73].

Assertions in GoldWorks are basically facts. There are two general types of assertions: structured and unstructured. Structured assertions are automatically entered in by GoldWorks whenever frames or instances of frames are created. Unstructured assertions are again divided into three

different types: regular assertions, relational assertions, and functional assertions. Regular assertions are just known facts entered into the knowledge base. Relational and functional assertions allow the developer to define a symbolic representation around an object and enter facts based upon this representation. For example, if the developer defines a relational symbolic representation around the object `floppy_disks` by the code in Figure 2-10, he can then make the assertions in Figure 2-11.

```
(define-relation floppy_disk
  (:relation-type: assertion)
  (size sided density bytes))
```

Figure 2-10: A Relational Symbolic Representation in GoldWorks

```
(floppy_disk 5.25 1 normal 100K)
(floppy_disk 5.25 2 double 360K)
(floppy_disk 5.25 2 quad 1.2M)
(floppy_disk 3.5 2 double 720K)
(floppy_disk 3.5 2 quad 1.4M).
```

Figure 2-11: User-Defined Assertions In GoldWorks

The major difference between the relational and functional assertions is that in a functional assertion, if an assertion already exists which matches the assertion being made with the exception of the last value, then the old assertion will be retracted (13:71).

Whenever the developer desires that GoldWorks query the user for additional information, he may use one of the three popup frames. A popup frame either 1) asks the user for confirmation by answering a yes-no question, 2) has the user type in the answer to a question, or 3) has the user make a choice among a list of choices (9:91).

Certainty factors in GoldWorks range from 0.0 to 1.0. A fact with a certainty factor of 1.0 is known to be absolutely true whereas something with a certainty factor of 0.0 is known to be absolutely false. The way certainty factors combine in rules is left up to the expert system developer. The certainty factor function provided with GoldWorks, which the developer may use, multiplies the lowest certainty factor of the antecedent by the certainty factor of the rule (29:184-185).

III. An Overview of ABC

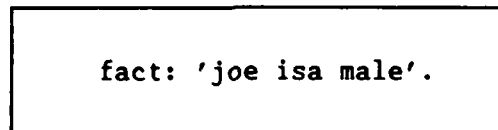
This chapter provides an overview of the ABC expert system shell. It will cover ABC's knowledge representations: facts, rules, askables, and frames. The chapter will touch upon how ABC deals with working memory, the dynamic part of the knowledge base, and will conclude with an introduction to the files which may be needed to run a consultation with ABC.

Knowledge Representations

This chapter will take a broad view of the term knowledge representations. Any structure used to represent either explicit domain-specific knowledge or implicit domain-control knowledge will be considered a knowledge representation. In ABC, there are four methods of representing knowledge: the fact, the rule, the askable, and the frame. The first three of these representations are derivations of similar representations used in BC3. The frame representation was one of the focal points of this thesis and will be discussed briefly in this chapter; the interested reader will find further details in Chapter VI. All four of these representations work with Object-Attribute-Value (OAV) triples.

Facts. Facts in ABC represent triples which are known to be true. They are used in much same the way that facts are used in M.1 or nonstructured assertables in GoldWorks. The fact that all fathers are males or all mothers are females can be represented as a fact in the development of a genealogical expert system using ABC. Almost any known truth which can be mapped into a OAV triple can be represented in ABC as a fact.

An example of how a fact may appear in the knowledge base is shown in Figure 3-1.



fact: 'joe isa male'.

Figure 3-1: An Example of a Fact in ABC

In this example, the OAV triple is "joe isa male." The triple used in this manner represents a known truth that joe is indeed a male. Facts such as this one could also be represented in BC3 using the same syntax.

Facts in ABC can also use certainty factors. Representation of facts with varying degrees or levels of certainty can be expressed. This is very important because many expert systems encode heuristic knowledge, or rules of thumb, into their facts and rules. Allowing for uncertainties in facts gives the expert system developer the flexibility to encode rules of thumb which call upon facts with multiple levels of truth values.

Briefly, certainty factors in ABC must be between 0 and 100. When a fact or rule is known with complete certainty, its certainty factor is 100. If it is known to be false, then its certainty factor is 0. The interested reader should read Chapter VII for additional information concerning certainty factors within ABC.

A developer writing an expert system for a hospital may wish to insert a fact that aspirin upsets the stomach. If it were known that aspirin upsets the stomachs of approximately 40 percent of the population, that fact may be represented in ABC as shown in Figure 3-2.



fact: 'aspirin upsets stomach cf 40'.

Figure 3-2: An Example of a Fact in ABC With Certainty Factor

The example in Figure 3-2 represents that aspirin will cause an upset stomach with a CF of 40. If it were further known that this expert system was going to be used in an emergency ward where it was critical to keep the likelihood of an upset stomach to a minimum, a certainty factor of 90 might be used.

If a fact is written without a certainty factor, as in Figure 3-1, the certainty factor is assumed to be 100. Certainty factors of zero are not normally used but facts with such certainty factors may in some cases make a knowledge base more readable or maintainable.

Rules. The rules in ABC are considered the most crucial representation of knowledge. Rules are somewhat analogous to the hub of a wheel where facts, askables, and frames are spokes which aid and support the rules. See Figure 3-3 for illustration.

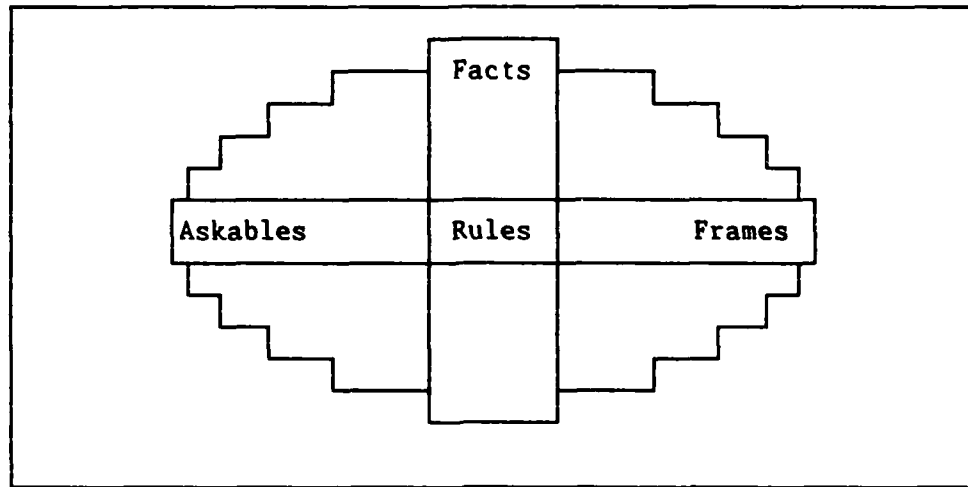


Figure 3-3 Representations Center Around Rules

Facts, askables, and frames are all similar in one respect; they all provide OAV triples in a representation such that the rules can use this knowledge to infer new knowledge. It is in the rules where heuristic knowledge can be represented and have its greatest effect.

An example of a rule in ABC, which could be part of an expert system to diagnose car faults, is seen in Figure 3-4.

This example represents the heuristic knowledge that would indicate the battery as the problem with a certainty factor of 85 if the car's engine wouldn't start and if both the headlights and the horn didn't operate.

```
rule_23: if    'engine will_not start' and  
              'headlights will_not operate' and  
              'horn will_not operate'  
            then 'problem is battery cf 85'.
```

Figure 3-4: Example of a Rule in ABC

Since ABC is a backward chaining expert system shell, a goal or subgoal must match or be instantiated to the conclusion of a rule in order for the rule to contribute to the solution. If the goal or subgoal in the rule shown above was 'problem is Fault', then starting from the first condition of the rule, each condition is made a subgoal and is checked to see if a solution exists for that condition. If all of the conditions of a rule can be solved, then the conclusion of the rule is said to have a solution. If any of the conditions fail, then the entire rule fails.

Askables. Askables in ABC represent OAV triples which need to be verified or chosen by the end-user. In its simplest form, an askable requires the end-user to answer either yes or no to a question. In its more complex form, an askable asks a question, provides an enumerated list of possible answers, and requires the end-user to select the best choice. In either case, an askable is called upon only as a last resort to solve an OAV triple for which no facts, rules, or frames provide information. The one exception to this rule is an askable called "initial-askable".

Initial-askables are askables which prompt the user for information automatically at the beginning of every consultation. An example of an initial_askable which might be used in a wine advisory expert system is shown in Figure 3-5.

```
initial_askable: 'main_component is Variable' derived from
                  "What is the main component of the meal?" and 'meat fish poultry'.
```

Figure 3-5: Example of Initial_Askable in ABC

This askable would be called at the beginning of a consultation with the wine advisory expert system and the user would see the ABC-generated prompt shown in Figure 3-6 displayed on his monitor.

What is the main component of the meal?

1. meat
2. fish
3. poultry

Enter Number or w (for why) >

Figure 3-6: An Example of an ABC Initial_Askable Prompt

After the end-user enters a valid number, the OAV triple is asserted into the Prolog database and is considered a fact from that point on. If the user were to select number two in the above example, the triple

"main_component is fish" would be asserted as a fact for the rest of the consultation.

Frames. Frames in ABC are, to a small degree, similar to facts. Frames either exist or they don't. If they do exist, they do so to support the rules. Frames differ from facts in at least two major points. Frames allow for the hierarchical structure of objects in a more efficient and easier-to-read format than facts. Frames also make use of demons, i.e., automatic execution of Prolog procedures, if their slots are designated by the developer with either an if-needed, if-added, or if-remove facet. An example of a frame in ABC is illustrated in Figure 3-7.

```
frame : cabernet_sauvignon
      slot 'color value red'
      slot 'body value light medium full'
      slot 'sweetness value dry medium'
      slot 'cost if_needed find_cost'.
```

Figure 3-7: An Example of a Frame in ABC

The frame represents the wine cabernet sauvignon. Each of the frame's slots, except the last one, is a "value" slot. Value slots hold one or more values but do not execute demons. The last slot, cost, is an if-needed slot. If the cost of this wine is needed, the demon procedure called "find_cost" will automatically be executed.

Working Memory

The knowledge base of ABC can be subdivided into two parts: static and dynamic. The static portion of the knowledge base contains rules, basic facts, and askables which were entered into the knowledge base by the expert system developer. Usually, the static knowledge of the expert system is entered before it is fielded or during a modification.

In comparison, the dynamic part of the knowledge base, called working memory, is knowledge which is gained dynamically from the use of the expert system itself. During a consultation with the end-user, an askable or demon procedure can extract knowledge from the end-user. This extracted knowledge can be saved so that in future consultations, the end-user will be spared from repeatedly entering the same information.

One major advantage of working memory is its ability to grow. With a well thought out static knowledge base, it is conceivable that a small working memory could grow to several times its original size.

In ABC, working memory is kept in a separate file from the rest of the knowledge base. By using separate files, working memory can be loaded into the Prolog database or saved from the Prolog database to a file without altering the static portion of the knowledge base.

IV. BC3 - Phase One: The Origin of ABC

This chapter has two main objectives: 1) to review the expert system shell BC3, and 2) to explain what inefficiencies were found in BC3 and what code modifications were made to correct them. In this chapter, the knowledge representations used in BC3 will be discussed in addition to the "How" and "Why" trace facilities. The last section concerning the first objective of this chapter will provide information on the user interface of BC3. The remainder of the chapter will detail two problems found in BC3 and what actions were taken to solve them.

Knowledge Representations

The knowledge representations in BC3 do not stray far from the equivalent underlying Prolog representations. In Prolog, there are facts and rules expressed with horn clauses. In BC3, there are also facts and rules. These facts and rules are expressed exactly the same way as the facts and rules covered in the last chapter on ABC with one exception. The facts and rules in BC3 can not provide a certainty factor. Basically, all knowledge in BC3 must be symbolically represented in the form of object-attribute-value (OAV) triples. Furthermore, these OAV triples must be in the form of a BC3 fact, rule, or askable, each of which is described below.

BC3's representation enables the inference mechanism to manipulate the knowledge in the knowledge base and create and maintain both the "how" and "why" trace.

Facts in BC3 are expressed in the form "fact : [obj,attr,val]." where the object-attribute-value (OAV) triple is being asserted as a given or a truth. This same fact could be represented in Prolog as "attr(obj,val)." For most beginning students, the BC3 form of the representation is much more "user friendly" and readable.

Likewise, students will find rules in BC3 are also more understandable than their Prolog counterparts. An example of a rule in BC3 is shown in Figure 4-1.

```
rule_23: if [obj1,attr1,val1] and
           [obj2,attr2,val2] or
           [obj3,attr3,val3]
         then [obj4,attr4,val4].
```

Figure 4-1: An Example of a Rule in BC3

The BC3 rule in Figure 4-1 is equivalent to the Prolog rule in Figure 4-2.

```
attr4(obj4,val4) :-
    attr1(obj1,val1),
    attr2(obj2,val2);
    attr3(obj3,val3).
```

Figure 4-2: Prolog Equivalent of BC3 Rule

Again, the BC3 rule provides the same basic functions as the underlying Prolog, yet with a more "English-like" structure at the moderate cost of some efficiency.

A BC3 askable does not have a straightforward Prolog equivalent representation such as a fact or rule. The askable in BC3 is merely a OAV triple which can take one of two forms based upon the value associated with the triple's object-attribute pair. The askable provides the inference mechanism with some other method of solving a goal: it asks the user for additional information concerning an OAV triple. If the "value" portion of the triple is instantiated to a single value, a Prolog atom, then the BC3 askable simply asks the user to verify the OAV triple. A second form is where the value is uninstantiated. Like the first form of askable, this form will prompt the user to select the correct value from a list of possible valid values.

An askable in BC3 can thus be seen to have the behavior which could be produced with the following Prolog code: `attr(obj,Value) :- ask_question(obj,attr,Value)` where the Prolog predicate `ask_question` would have to determine which of the two forms of askable is required and provide the necessary question to the user.

The How Trace

There exist problem domains which, if an expert system was developed which could provide solutions to these problems, it would be almost as important to know how the solution was derived as opposed to knowing the solution itself. To be able to tell the user how a solution was derived, BC3 creates and maintains a "how" trace. The "how" trace is activated at the user's request; the user is provided a path between the goal and the solution which shows each fact, rule, or askable which was necessary

to satisfy the goal. The method BC3 uses to accomplish this task takes advantage of Prolog lists.

Unlike ABC, where the trace is kept in the Prolog database under the predicate 'trace/1', BC3 keeps a trace in the form of an accumulating list. This list is initialized in the BC3 predicate 'solve/3' and grows each time it gets passed to one of BC3's 'is_known/3' predicates where a triple is found to be true. When a solution is reached during a consultation, the goals are placed at the beginning of the list and the user is prompted to see if a trace of the solution is needed. If the user requests a trace, the final "how" trace list is processed through a "pretty" printing procedure. The output of this procedure is sent to the display monitor to show the user the path taken between the goals and the solution.

The Why Trace

BC3 allows its inference engine to prompt the user for the value of an object-attribute pair, via an askable, if the value cannot otherwise be found. When this happens, BC3 allows the user to ask why the inference engine is prompting for this particular value. To explain why a prompt is being asked, BC3 maintains a "why" trace. BC3's "why" trace works with Prolog lists in a fashion similar to the "how" trace, but the list is not passed around as an argument of a predicate.

Like the "how" trace, the "why" trace is initialized with the BC3 predicate "solve/3." The "why" trace is reinitialized for each goal by placing the goal itself in the "why" trace list. This list is stored in the Prolog database under the predicate "why_trace/1." The "why" trace

list gets modified if a rule is used to solve the current goal. The rule along with all of its conditions gets placed into the list and the list gets reasserted into the Prolog database. When the inference engine now resorts to the last attempt effort to solve a goal or subgoal via an askable, the user may reply to a BC3 generated question by replying "why."

When the user does respond to a question with "why", BC3's predicate 'explain_why/1' retracts the "why" trace list from the Prolog database and displays it. After the trace list shows the inferencing steps leading up to its question, the question is asked again.

User Interface

There are two types of users of any expert system shell. There is the developer, who uses the expert system shell to produce the expert system, and there is the end-user, who uses the resulting expert system to solve a class of problems. More is said about the two types of users in Chapter V. There can therefore be at least two types of user interfaces: one for the developer and one for the end-user. This is true for the commercial expert system shell GoldWorks.

In BC3, the developer must enter knowledge, in the form of facts, rules, and askables, into an ASCII file via a text editor to generate an expert system knowledge base (KB). Previous sections of this chapter provide an example of how both a fact and a rule are represented in BC3. An askable can be presented in a similar fashion. An example of an askable in BC3 is illustrated in Figure 4-3.

```
askable: [Student,sex,[male,female]].
```

Figure 4-3: An Example of an Askable in BC3

Because the user interface between BC3 and the developer uses the Prolog read predicate, each fact, rule, or askable read from the developer's knowledge base must be a proper Prolog structure. Since the structure used in BC3 to hold the OAV triples is the Prolog list, the simplest solution to the interface problem was to require the developer to enter OAV triples in the form of a Prolog list (see Figures 4-1 and 4-3). Similar associations between BC3 and Prolog can be seen by the end-user because of the end-user interface.

Whenever the end-user is prompted to answer questions from either control mechanisms or askables, his answer must be followed by a period and then followed by a return. The Prolog reader imposes this format-restraint in order to read Prolog terms from the input device. An example of this is when BC3 asks "Do you wish to see how this answer was arrived at?". The user, wanting to respond negatively, must respond with "n.<CR>" where <CR> signifies the depressing of the carriage-return key.

BC3 Problems and Solutions

The first of the four phases of this thesis was to study BC3, understand how and why it functions, and attempt to improve its effectiveness or efficiency without changing its function or user-

interfaces. There were two problems uncovered during this phase: 1) The ordering of the 'is_known' predicate, which controlled the inferencing in BC3, and 2) the handling of the "how" and "why" trace mechanisms. Both of these two problem areas were investigated using Arity Prolog's trace and timing utilities. The problems and their solutions are described below.

The 'is known' Predicate. BC3 builds upon Prolog's basic backtracking inferencing mechanism to provide its own inferencing. It accomplishes this task through the use of a predicate called "is_known". The efficiency of BC3's is_known predicate was found to be a problem.

In order to better understand the problem with the is_known predicate, some additional background is needed. In a backward-chaining expert system shell, the goal is known and the inference engine attempts to see if the knowledge base can confirm the goal. When a goal or subgoal, represented by an OAV triple, is trying to be confirmed in BC3, the inference mechanism goes through a series of checks.

The steps BC3 goes through to see if an OAV triple is known are outlined in Figure 4-4.

To increase efficiency and reduce unnecessary backtracking, one of the solutions was to reduce and reorder the rules within the is_known predicate. Five of the rules, which correspond to steps 2, 4, 6, 8, and 11 in Figure 4-4, all look into the trace to see if a triple had been solved previously. One rule, more general in its search pattern, replaced all five of these rules while maintaining the integrity of the original five rules.

- 1: It checks to see if the triple had previously been denied.
- 2: It checks to see if the triple is a fact and had been solved before.
- 3: It checks to see if the triple is a fact in the knowledge base.
- 4: It checks to see if the triple was confirmed by the user and had been solved before.
- 5: It checks to see if the triple was confirmed by the user.
- 6: It checks to see if the triple is a Prolog goal which had been solved before.
- 7: It checks to see if the triple is a Prolog goal which can be solved.
- 8: It checks to see if the triple is a rule which had been solved before.
- 9: It checks to see if the triple is a rule whose conditions are known.
- 10: It checks to see if the triple is part of a rule structure, ANDing, ORing, or NOTting it to additional conditions and recursively tries to solve the other conditions.
- 11: It checks to see if the triple was told by the user and was solved before.
- 12: It checks to see if the triple is "askable" and if possible, asks the user to either confirm the triple or provide the missing link of the triple. These steps are controlled by the 17 rules which make up the is_known predicate.

Figure 4-4: BC3's 'is_known /3' Inferencing Steps

To maximize efficiency, the rules with the most specific input requirements should always go first, assuming that the logic is not changed. By reducing the rules within the is_known predicate and by reorganizing the remaining rules by placing the rules with the most

specific arguments before the more general rules, a 38.8% reduction of time was realized to run an expert system consultation (refer to last section of this chapter).

Altering The Trace Mechanisms. The second method found to increase both the efficiency and the effectiveness of BC3 was to alter both the "how" trace and the "why" trace. As mentioned previously, the how trace is passed along as a list during the recursive calls to the 'is_known' predicate. List manipulations, such as 'append' and 'remove', make both trace mechanisms highly inefficient in large knowledge bases where lists can get lengthy.

A relatively simple solution to this problem was to assert each step of the solution into the Prolog database under a cover predicate called 'trace'. This not only makes BC3 much faster, up to 35.5% faster (refer to last section of this chapter) with a small knowledge base, but also frees stack space. Depending on the interpreter being used, stack space could be a precious commodity easily depleted by anything other than a small knowledge base. To view the trace, a match-write-fail loop steps through the database sequentially displaying the trace in proper sequence.

Testing Solutions. Using two modified versions of a knowledge base, called "ttl2.bc3" and "ttl3.bc3", developed by the author in the introductory AI class, EENG-592, a series of five tests was performed. Each of the tests was run on a IBM AT running at a clock speed of 6 megahertz. The Prolog-1 interpreter was used on the first test and the Arity version 5.0x Prolog interpreter on the remaining tests. The knowledge base "ttl2.kb" was designed to succeed after traveling through

a search path which consisted of approximately two dozen rules. The knowledge base "ttl3.kb" was identical to "ttl2.kb", but was given a different goal such that it could not succeed and would fail after trying to solve only seven or eight rules.

The first test-set was made using the unmodified BC3 with the Prolog-1 interpreter. The times were arrived at using a stop watch and was truncated to the nearest second. The remainder of the tests were made using the Arity interpreter which made use of built-in predicates to perform timing tests to within one one-hundredth of a second. The second test set was made using the unmodified BC3. The third test set was made using the new trace mechanisms mentioned above. The fourth test set was made using the both the new trace mechanisms and the new rule arrangement within the is_known predicate. The last test set was made using the ABC shell after the second phase and is entered here for the curious reader. All tests except those in the first set were completed using the Arity interpreter. The test results are shown in Figure 4-5.

Expert System Shell Description	Time for ttl2.bc3	Time for ttl3.bc3
BC3.PRO (using Prolog-1 - BASELINE)	1' 52"	34"
BC3a.ARI (same only using Arity)	31.74"	7.75"
BC3b.ARI (new trace mechanism)	26.58"	5.72"
BC3c.ARI (new is_known arrangement)	16.26"	5.38"
ABC_p2.ARI (phase 2 of ABC)	14.17"	5.87"

Figure 4-5: BC3 Test Times

V. Phase Two: ABC's User Interface

One of the primary objectives of this thesis was to provide a better user interface to an educational expert system shell than was previously available with BC3. This task was designated as the second phase of a four phase approach in the realization of ABC. The primary concern was to isolate the user from the constraints that Prolog imposes when interfacing with ABC. Included in this chapter is discussion on the different approaches which were considered relevant to a user interface, along with the solution chosen and some of the mechanics involved.

The Two Types of Users

As was stated in Chapter IV, there are at least two types of users in every expert system shell: expert system developers and end-users. The developer acts as a "knowledge engineer," encoding domain-specific knowledge into the rules and facts of a knowledge base. The end-user is anyone who takes advantage of the expertise in the expert system to perform a particular task. The developers are generally more knowledgeable about the use of facts and rules and programming in general than the end users. Still, it is important to make the developer's interface as "friendly" as possible without sacrificing too much flexibility, efficiency, or power.

The goals of the developer and the end-user are very different. The developer wants to input test cases to make sure the knowledge base is complete or "robust." He may wish to add, delete, or modify existing

facts or rules on-line to study the effects to insure this completeness (22:135-139). Usually, the end user is not concerned with such matters.

The end-user is more concerned about how easy it is to get information about his problem entered into the expert system in order to get a solution which he can understand. If the expert system prompts him for a question, he doesn't want the question to be cryptic nor does he want it to be ambiguous. When replying with an answer, end users would usually prefer to make one simple and straightforward keystroke as opposed to a sequence of keystrokes requiring memorization.

In an educational expert system shell, the developer and the end user are frequently the same. Since an educational expert system shell is designed for students, it is important to have a good user interface at both the developer's level and for the end user.

The Command Line

Execution within the top-level of BC3 is very sequential. Upon starting BC3, a sequence of events take place: 1) the shell prompts the user for the name of the knowledge base file, 2) working memory, if present, is consulted into the Prolog database, 3) the consultation begins, 4) the consultation ends, 5) the user requests a trace, 6) working memory is saved. This sequence cannot be altered.

One of the first changes to the user interface of BC3 was the adoption of a command line. The new command line is similar to Teknowledge's M.1 command line. It modularizes the top level into functional divisions. Each ABC command corresponds to one of the functional divisions. The "how" trace is an example of one of these

divisions. Typing "trace" at the ABC prompt will provide the end user with the most current "how" trace and return him to the ABC prompt. At any ABC prompt, the user may type any of the ABC commands (see appendix C for a list of ABC commands).

The Developer's Interface

With BC3, the developer enters facts and rules into an ASCII file using the symbolic object-attribute-value (OAV) triples to encode the domain knowledge. In BC3, the OAV triples have to be represented in the form of a Prolog list (see Figure 5-1). Several methods were investigated to read the OAV triples into the Prolog database without having the triples placed into a list structure imposed by the Prolog reader.

All of the methods researched to read in OAV triples were in one of two categories: methods using the Prolog reader or methods using the 'get0' Prolog predicate. The procedures tested using the Prolog reader to read OAV triples from a file were anywhere between one and two orders of magnitude quicker than a procedure using the 'get0' predicate. When using the 'get0' predicate, each character of a file (printable or not) has to be checked to see if it marks the end of a file or the beginning of a comment. This extra overhead makes the 'get0' predicate inefficient when used to read large files.

There are a couple of published examples of standard Clocksin and Mellish Prolog procedures which read a structure, such as a normal English sentence, and parse it into words (5:102-104, 15:203-210, 3:148-151). Modified versions of these procedures showed their usefulness in

parsing small sentences of a very narrowly defined structure (i.e., words separated by spaces). However, when these procedures were modified to check for the end of file or the beginning of a comment, or were modified to accept both Prolog code and some other predetermined structure, they were annoyingly inefficient.

Using the Prolog reader not only provided superior efficiency, but also allowed the use of Prolog comments in the knowledge base file without having to be concerned about their entering the Prolog database. Also, whenever any atom starting with an underscore or capital letter is read by the Prolog reader, it is immediately stored in the Prolog database as a variable.

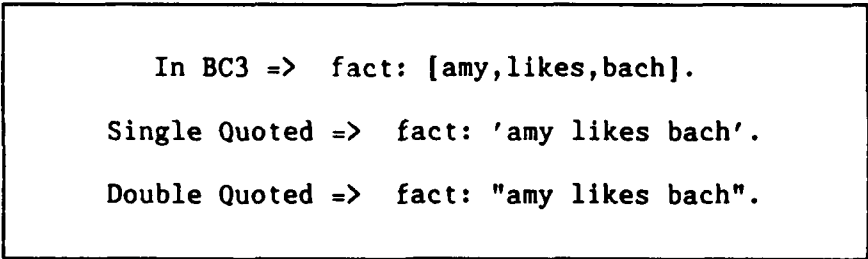
There are however, some drawbacks to using the Prolog reader. To input structures via the Prolog reader requires that the structures be in proper Prolog format.

Single Quotes: An Alternative to Prolog Lists. Deciding on using the Prolog reader when faced with the only alternative, the Prolog 'get0' predicate, was no hard decision. Tests showed that while the procedures using the Prolog 'read' predicate were not as fast as procedures using other non-standard Prolog predicates (available with the Arity interpreter), they were practical for pedagogical purposes.

To rid the knowledge base of OAV triples wrapped in Prolog lists, two procedures were considered using the Prolog reader. The first procedure used single quotes to transform the OAV triple into an atom. The second method considered used double quotes. The second method turned the OAV triple into a list of ASCII numbers. The code was the same for both procedures with one exception. When using single quotes,

the OAV triple, now an atom, had to be transformed into an ASCII list. The procedure using double quotes did this automatically.

The procedure using single quotes was chosen as being better for two reasons. Studying knowledge bases which had both single quoted and double quoted OAV triple representations, the single quoted OAV triple looked clearer, more appealing, and easier to read. Secondly, since both procedures are rather complex, the time it took to transform an atom to an ASCII list using the Prolog predicate 'name' could be treated as negligible. Figure 5-1 illustrates how a fact is entered in the knowledge base file in BC3 along with how it is entered using both the single quoted and double quoted procedures.



```
In BC3 => fact: [amy,likes,bach].  
Single Quoted => fact: 'amy likes bach'.  
Double Quoted => fact: "amy likes bach".
```

Figure 5-1: Different Ways to Enter OAV Triples

ABC Commands For The Developer. There are several ABC commands provided to make the developer's job easier. The developer, or the end user, can review, add, or delete goals online. He may also review frames, review rules, or add a frame from inside ABC using the appropriate commands. See appendix C for complete details on all the commands available in ABC.

The Mechanics Involved. The entire procedure used to read in a knowledge base full of facts, rules, and askables in ABC is simple for the user to accomplish.

First, the user must tell ABC that he wants to load a knowledge base. This is accomplished by the user's typing "load" at the ABC prompt. This will in turn invoke ABC to prompt the user to supply the filename for the knowledge base. This prompt is shown in Figure 5-2. The user types in the filename, with extension of either ".kb" or ".abc" (more on the extensions later) and ABC will respond by doing one of two

```
ABC > load
```

```
Enter the name of the file where your knowledge base is stored,  
or enter <Return> to abort.
```

```
Filename, including path is:  vine.abc
```

Figure 5-2: The Load Prompt in ABC

things. Either ABC will read and parse the knowledge base with extension ".kb", create the knowledge base with the ".abc" extension then read the ".abc" file or it will simply read in the knowledge base with extension ".abc". The decision is invisible to the user; ABC will parse the file name and automatically decide which of the two courses of action is appropriate.

The knowledge base with the ".kb" extension represents OAV triples as atoms within single quotes. The expert system developer would

normally use a text editor and create and save a knowledge base to an ASCII file with the ".kb" extension. However, the internal symbolic representation of OAV triples in ABC, like its predecessor BC3, are Prolog lists. In order for the original knowledge base with the ".kb" extension to be of any use, it must successfully be converted. This conversion process is briefly discussed in the following paragraph.

First, a term is read from the knowledge base file. This term is then parsed into legal components such as triples or operators. If the component is a triple, then the triple is converted into a list. Once all of the triples have been converted into list structures, the converted term is written out to the new file with the ".abc" extension. This process continues until all the terms in the original knowledge base has been read, converted, and written to the new file. After the new file has all the parsed and converted knowledge written to it, it is then closed as an output file and immediately reopened as an input file and all the terms are read again, only this time all the OAV triples are in Prolog list format. The new terms are then asserted into the Prolog database.

Subsequent loading of the knowledge base is much faster if the end user loads the parsed and converted knowledge base with the ".abc" extension.

The End-User's Interface

In BC3, there are two problems with the end-user's interface. The first problem has two parts which were corrected with two solutions. When BC3 prompts the user with a yes-or-no type question, the user has

to reply with at least a "y" or "n" followed by a period and then followed by a carriage return. The second part of the first problem is similar. When BC3 prompts the user with a question which requires the user to make a selection from a list of valid answers, the user has to enter the exact answer, with no misspelling, followed by a period and then followed by a return. The second problem was the way questions were posed to the end-user. BC3 did not allow an "askable" to have a unique style of question. Each question would be asked in a fashion similar to Figure 5-5.

'get_reply' and 'readline' Predicates. To solve the first problem, two predicates were formed: 'get_reply' and 'readline' (see Figures 5-3 and 5-4). The 'get_reply' predicate simply got the first character which was checked to see if it was either a "y" for yes, a "n" for no, a "w" for why, or just a carriage return which always defaulted to mean yes. Rather than using the 'read' predicate after a yes-or-no type question, which requires at least three keystrokes, the new 'get_reply' predicate gets the user's response with one keystroke. If the response is invalid, the computer will alert the user and prompt for another answer.

The 'readline' predicate allows the user to type in letters, symbols, and numbers into a buffer while they simultaneously show up on the computer's screen. Using the rubout key is also permitted, to delete existing characters. The 'readline' predicate is transparent; the end-user is not made aware that control of what is being presented on the screen has been changed. When the return key is pressed, the contents of the buffer are sent back as a Prolog atom-type argument. This predicate

```

get_reply(Reply) :-
    get0(User_Reply), nl,
    ( User_Reply = 13,
      Reply = yes
    ;
      User_Reply = 121,
      Reply = yes
    ;
      User_Reply = 110,
      Reply = no
    ;
      User_Reply = 119,
      Reply = why
    ;
      nl,nl,
      write('You must enter either a "y" or "n", or if you wish, '),
      nl,write('just hit a return for yes. '),
      get_reply(Reply)
    ).

```

Figure 5-3: The ABC 'get_reply' Predicate

allows data to be entered into ABC without the need for Prolog structures or the period which is required by the Prolog reader.

Enumerated Askables. Whenever the user is prompted by an askable-generated question, BC3 will provide a prompt similar to the one in Figure 5-5. This prompt implies that the value for the object-attribute pair is not known and that with the data in the knowledge base, it will never be known. It requests the user to select the appropriate answer from a list of valid or legal values and to type that value in.

ABC corrects this user interface problem by placing a question inside each askable. This allows the developer to create a question tailor-made for the situation and does not transfer the burden of understanding what is being asked to the end-user.

```

readline(Temp_List, Line) :-
  get0(Char), !,
  ( Char = 13, !, nl,          /* A Return Key is */
    reverse(Temp_List, Line_List), /* Pressed. */
    name(Line, Line_List)
  ;
    Char = 8, !,              /* The Rubout Key */
    put(32), put(8),          /* is Pressed. */
    Temp_List = [Head|Tail],
    readline(Tail, Line)
  ;
    identifier(Char), !,      /* Char is a legal */
    readline([Char|Temp_List], Line) /* identified char- */
  ;                             /* character or a */
    Char = 32, !,             /* space. */
    readline([32|Temp_List], Line)
  ;
    readline(Temp_List, Line)
  ), !.

```

Figure 5-4: The ABC 'readline' Predicate

```

main component is?
Legal values: [meat,fish,poultry]
> meat. <CR>

```

Figure 5-5: Prompt From BC3 Askable

Using ABC, all enumerated askables are displayed as shown in Figure 5-6. The askable in Figure 5-5 is the same as the one used in Figure 5-6. Notice the typical reply which is required by BC3 is less intuitive than the reply required by ABC (printed in bold characters).

The user-interface was tested using the wine knowledge base in Appendix-G and was successful in isolating the user from the underlying Prolog reader constraint of entering replies in Prolog syntax.

What is the main component of the meal?

1. meat
2. fish
3. poultry

Enter Number > 1

Figure 5-6: Prompt From ABC Askable

Summary

The developer's and end-user's interfaces have been enhanced over what was currently available with BC3. The developer does not have to type in OAV triples in the form of a Prolog list and the end-user is prompted by specific customized questions. Additionally, valid answers to questions are enumerated, enabling the end-user to select an answer with a single keystroke.

VI. Phase Three: Implementing Frames

The primary purpose of this thesis was to integrate Object-Attribute-Value triples with frames. This meant that a frame structure had to be found or developed which would allow such integration. This chapter will cover the frame structure which ABC uses, how this structure allows the integration of OAV triples, and will briefly discuss the underlying frame language which allows it to perform. The task of integrating frames into ABC was designated as phase three of a four phase approach to its completion.

The Frame Structure

The basic frame structure in ABC uses the frame itself to represent an object. Since an object may have one or more attributes, each attribute having one or more values, the frame structure must be able to symbolize an object's attributes and values efficiently.

Frames can have one or more slots. The basic structure of ABC frames makes use of this characteristic by using slots to represent the object's attributes. This allows an object being represented with a frame to have as many attributes as necessary to define it to the level of abstraction or detail needed. An object-attribute pair can therefore be represented by a frame, whose frame-name is the object, and having at least one slot whose slot-name is the attribute.

For the ABC frame structure to work with OAV triples, the value of a frame slot must represent all of the possible values of an object-attribute pair. This is accomplished in ABC by placing the object-attribute values in a list and having this list be the slot value.

Each frame in ABC is also capable of supporting demons. Demons in ABC are Prolog goals which may be procedural in nature. They are automatically executed whenever the inference engine attempts either to get a value, add a value, or delete a value from a frame; the slot's facet is either "if-needed", "if-added", or "if-removed" respectively. For additional information on how demons function, refer to the user's manual in Appendix C.

The basic structure of the ABC frame is illustrated in Figure 6-1. ABC integrates frames and OAV triples as follows: a frame is the object of an OAV triple; the frame's slots are the attributes of the object represented by the frame, and the value of each slot is a list which represents the values of the object-attribute pairs.

The Frame-Base Language

Once the conceptual part of developing the frame structure was complete, the next step required Prolog code which would tie frames together with the prototype ABC system developed from the first two phases. Rather than developing the code using a top-down approach, the framebase language was developed from the bottom up.

The basic structure of a frame in ABC appears similar internally to the frame shown in Figure 6-2. The frame in Figure 6-2 has slots which

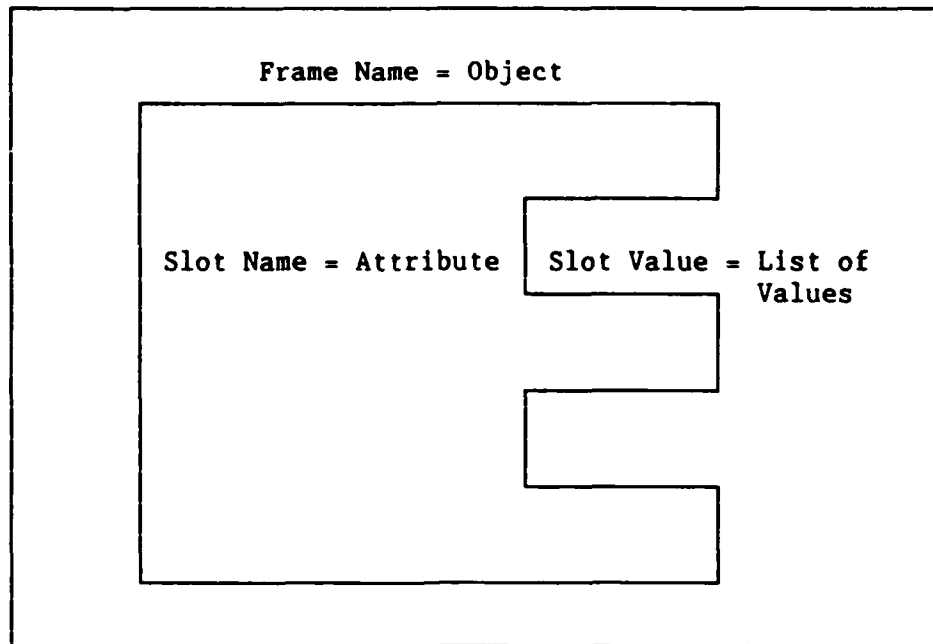


Figure 6-1: ABC's Frame Structure

are made up of a list containing the slot name, the slot facet, and the slot values. The first slot in the figure indicates the attribute as "ako", which is a synonym for "a kind of." These attributes allow frames in ABC to acquire attributes and values from frames higher in their hierarchical tree. The last two slots are examples of how demons are placed within a frame.

```

frame : tweetie
  slot [ako,value,canary]
  slot [owner,value,joe,mary]
  slot [born,if_needed,ask_dob]
  slot [age,if_needed,find_age].

```

Figure 6-2: A Typical ABC Frame

A frame-base language was then written based upon three Prolog predicates: 'frame_get', 'frame_put', and 'frame_delete'. Much of the Prolog code was inspired or derived from previous work in a similar area (6:237-258).

Accessing a Frame Value. Obtaining the values within a frame slot is fundamental in any frame language. Given a frame name and a slot name, a Prolog procedure was written called 'frame_get' to obtain a list of all values for that particular slot. To obtain an individual value, the calling procedure can use the standard 'member' predicate on the list returned from the 'frame_get' predicate.

The 'frame_get' predicate is hierarchical. It will first look for a frame slot with the facet being set to "value". If the value slot cannot be found or the value slot does not contain the necessary value, the 'frame_get' predicate will continue to search. The 'frame_get' predicate will next search for a slot with the facet being set to 'default'. If either the default slot does not exist or the value needed is not in the default slot then the search continues. Lastly, 'frame_get' will search for a slot which contains the 'if_needed' facet. The 'if_needed' facet will cause a Prolog procedure to be executed in order to obtain the necessary value. If this slot does not exist, or the demon called into execution by this slot does not succeed, then the 'frame_get' predicate will fail.

Adding a Value to a Slot. Adding a value to a slot makes use of an ABC predicate called 'frame_put'. When adding a value to a slot, 'frame_put' checks to see if a slot already exists which matches the

triple's attribute and has an 'if_added' facet. If there is such a slot, the demon procedure associated with the 'if_added' facet is executed to produce some side effect.

Regardless of whether there is a demon executed, the 'frame_put' procedure places a value in the appropriate slot of the frame, even if it means having to create a slot specifically for the value which is being added. If the appropriate slot already exists, the list of old values is retracted from working memory and the new value is appended to the front of the old list and reasserted into working memory. If the end user elects to save the working memory after a consultation, then the new values are asserted into the working memory file.

Deleting a Value From a Slot. A value may be deleted from a frame using ABC's 'frame_remove' predicate. If there exists a slot in the frame which matches the triple's attribute and furthermore has a 'if_removed' facet associated with it, then similar to the 'if_added' demon, the 'if_removed' demon would be executed, possibly providing some side effect. Again, regardless of whether an 'if_added' facet exists, the value, if it exists, will be deleted from the frame.

ABC's 'if_removed' predicate is smart enough to determine if the value is the last value in a particular slot. If the last value is removed from a slot, the slot itself is then removed from the frame automatically. Additionally, if the last slot is removed from a frame, the entire frame is deleted.

Adding or Deleting Slots or Frames. There are no predicates in ABC which will add or delete entire slots or frames. Although these predicates could be created easily, they are not necessary since other more general or higher-level procedures could be accomplished utilizing the three predicates, 'frame_get', 'frame_put', and 'frame_remove' along with other predicates such as 'retract_all'.

Summary

Integrating frames with OAV triples in ABC was accomplished by allowing the frame-name to be the object, the slot-name to be the attribute, and the slot-value to be a list of values which can be associated with an object-attribute pair.

VII. Phase Four: Uncertainties

This chapter covers how uncertainties are integrated into ABC. It is the last phase of a four phase development effort to create the ABC expert system shell. In this chapter, a brief review of how rules and facts use certainty factors is followed by a discussion of the search strategy which the inference engine uses to attain its goal. The last section will explain what modifications to the ABC "how" and "why" trace were required when uncertainties were added.

Uncertainties in Rules and Facts

Chapters I and II provide an explanation as to why it is so important to have the capability to reason with inexact knowledge via rules and facts employing some type of confidence mechanism. Briefly, the power to reason with rules and facts, representing expert domain knowledge with varying degrees of certainty, significantly expands the feasibility of expert systems in domains where human experts call upon their inexact or uncertain knowledge to make decisions (11:239).

Chapter III provides information on how uncertainties are placed into the rules and facts of ABC. When a fact has a certainty factor of 100, the OAV triple within the fact "is known" to be true with a certainty factor of 100. Unlike a fact with a certainty factor of 100, if a rule either implicitly or explicitly has a certainty factor of 100, the triple which is the conclusion of the rule is not necessarily "known" with a certainty factor of 100.

Certainty factors in rules follow five rules outlined in Figure 7-1. The OAV triple which is the conclusion of a rule "is known" if the OAV triples which make up the premise of the rule are known. The certainty with which an OAV triple is known if the rule succeeds is the product of the certainty factor of the rule's conclusion and the overall certainty factor of its premise.

1. The certainty factor of two or more conjunctive conditions in the premise of a rule is equal to the lowest certainty factor of those conditions.
2. The certainty factor of two disjunctive conditions in the premise of a rule is equivalent to the certainty factor obtained by viewing the disjunct conditions as two distinct rules.
3. The overall certainty factor of a rule is the product of the overall certainty factor of the premise and the certainty factor of the rule.
4. The certainty factor of two rules having the same conclusion can be calculated as follows.
$$CF(\text{Overall}) = CF1 + CF2 * (100 - CF1)/100$$
5. The certainty factor of an OAV triple asserted as "known" can be increased using the same calculation as shown in rule 4 above when additional rules or facts add support to the triple's validity.

Figure 7-1: Rules Concerning Calculating Certainties in ABC

ABC has basically the same certainty factor characteristics as are found in M.1 (Refer to Chapter II for overview of M.1). The predicate in ABC which calculates the resultant certainty factor when a previously asserted OAV triple is solved again, is the 'calculate_CF' predicate shown in Figure 7-2. Note that the 'calculate_CF' predicate implements rule four of the certainty factor rules listed in Figure 7-1.

```
calculate_CF(100,_,100).  
calculate_CF(_,100,100).  
calculate_CF(CF1, CF2, CF) :-  
    CF is CF1 + (100 - CF1) / 100 * CF2, !.
```

Figure 7-2: ABC's Calculate_CF Predicate

The Search Strategy

The educational expert system shell BC3 does not use certainty factors. Given a goal to prove, BC3 searches through the knowledge base using a directed depth-first search. BC3 proves its goal(s) either directly, by matching the OAV goal with an asserted OAV triple, or more commonly, indirectly, by finding a rule whose conclusion matches the goal's OAV triple and whose premise can be proven. Once a singular path is found, BC3 discontinues any further search for additional paths even if other solutions exist. In ABC, the search is more complex but the method used is comparable.

In ABC, unless the top goal is known with a certainty of 100, the search for additional solutions continue. The search for multiple solutions to a goal is performed using a directed depth-first search much like that used in BC3. The major difference between BC3 and ABC, with respect to its search for solutions, is that ABC will assert the solution as being found, check the solution to see if its certainty factor is less than 100, and if so, continue the search for a different solution or a different path to the same solution.

ABC terminates its search for additional solutions to a goal upon reaching one of two states: it finds a solution with a certainty factor of 100, or it exhausts all possible paths which could create additional solutions.

ABC takes advantage of Prolog's built-in backtracking capability to implement its directed depth-first search strategy. While this may not have been the most efficient search strategy, it was seen as the best method to approach such a formidable task and at the same time, provide an educational example of depth-first search.

The search strategy does impose a constraint on the expert system developer of which he should be made aware. An expert system will run more efficiently if facts or rules with certainty factors of 100 are placed before facts and rules, concerning the same OAV triple, with a lesser certainty factor. For example, Figure 7-3 shows the correct ordering of two rules. This constraint is imposed because of the strong reliance of ABC's inference mechanism on the the procedural aspect of the underlying Prolog control strategy.

```
rule_24 : if 'aspirin upsets stomach' and
           'patient complains about stomach_pain'
           then 'prescribe pain_reliever aspirin_free'.

rule_25 : if 'patient has liver_problem'
           then 'prescribe pain_reliever aspirin_free cf 85'.
```

Figure 7-3: Proper Ordering of Rules in ABC

Even though the first rule in Figure 7-3 doesn't explicitly declare its certainty factor, the default in ABC for all undeclared certainty factors is 100, thus; the rule with the certainty factor of 100 is placed before rules with the same conclusion with a lesser certainty factor.

Altering the Trace

Until this last phase, the trace mechanisms were relatively simple. Whenever an OAV was solved, it was asserted into the Prolog database along with how it was solved. When the end-user requested a trace output, the asserted OAV triples were retracted in reverse sequence from that in which they were asserted. Retracted along with OAV triples were the reasons for their assertions. The OAV triples along with their reasons for their assertion were processed through a simple procedure and an output was displayed which indicated the path between the solution and the goal.

Using certainty factors, the "how" trace had to be altered dramatically. Because ABC uses its "how" trace as a cache to determine if OAV triples have already been previously solved, duplicates of asserted OAV triples had to be prevented. Without preventing duplicates in the "how" trace, an OAV triple could be asserted more than once, each assertion with a different certainty factor, providing erroneous conclusions to a goal.

More importantly, when the inferencing engine of ABC was adapted for inexact reasoning, it acquired the new responsibility of producing multiple solutions and thus multiple solution paths. The "how" trace mechanism had to be altered to ensure that only those OAV triples which were actually in a valid solution path were retained in the trace. The "how" trace is managed by two predicates: 'assert_in_trace' and 'clean_up_trace'.

The 'assert_in_trace' predicate kept the "how" trace free of redundant OAV triples by checking for existence of triples prior to inserting them into the trace. When OAV triples were found to already exist in the trace, the 'assert_in_trace' recalculated the certainty factor of the previous asserted triple and replaced it with the new certainty factor. This certainty factor replacement was accomplished through the use of the 'replace_trace' and 'circulate_trace' predicates.

The 'clean_up_trace' predicate has the function of discarding all OAV assertions in the trace which do not ultimately lead to a successful solution. It accomplishes this task by checking each assertion in the trace and making sure that the assertion made before it is either a goal or relevant to the current assertion.

VIII. Testing ABC

This chapter will discuss how ABC was tested. The first section explains how the testing of individual predicates, the hierarchical building blocks of ABC, was implemented. The following section covers the results of testing ABC against prewritten knowledge bases.

Testing ABC Predicates. ABC comprises approximately seven dozen Prolog predicates. All but a half dozen of these predicates had to be uniquely coded for ABC. Whenever a small group of related predicates was written, a series of tests was run to see if each of the predicates operated within the constraints of its design. These tests varied widely, mainly depending on the complexity of the predicates' function.

Each predicate, upon passing its tests, was documented with a description of its function along with any constraints imposed upon the predicate or its arguments. Refer to appendices B, D, or E for specific information on any predicate in ABC. Each predicate which has an arity greater than zero has an argument constraint parameter associated with each of its arguments. These argument constraint parameters follow an easy-to-understand notation.

The constraint parameter notation was needed to provide information about whether a predicate's arguments are required to be instantiated or not instantiated. No industry standard seems to exist for such a notation, yet there is fair acceptance by Prolog programmers using Arity Prolog or Quintus Prolog on the notation illustrated in Figure 8-1; as an example, this notation is applied to the predicates 'execute', 'member', and 'get_rest_word'.

```
execute(+)  
member(?,?)  
get_rest_word(+,+,--)
```

Figure 8-1: Prolog Predicate Notation

The meaning of the notation in Figure 8-1 is straightforward. The "plus" symbol, representing the single argument in the 'execute' predicate, indicates that the argument has to be instantiated in order for the predicate to work correctly. In a like manner, a "negative" symbol represents that the argument is required to be an uninstantiated variable. The third symbol, the "question mark", indicates that the argument can be either instantiated or not instantiated and would work correctly under both conditions.

There are several other constraints which may be imposed upon a predicate's argument. Whenever a predicate requires its argument to be a list or an integer, this requirement will be listed in the source code documentation as a constraint.

Constraints which are not complied with may result in erratic behavior or system error messages. ABC predicates were coded wherever possible to prevent a system error or erratic behavior when faulty information is entered by the user. However, standard Prolog does not have the flexibility of Ada or other such languages to handle error exceptions (Standard Prolog predicates are listed in Appendix I).

An example of when a system error would occur is when ABC prompts the user for either a filename, or for whether the working memory or auxiliary files should be loaded. If the specific file cannot be found, a system error will be generated. Standard Prolog does not have the facilities to prevent this type of error.

Testing ABC Using Prewritten Knowledge Bases

As previously mentioned, the specification of ABC takes the form of a user's manual. To supplement the requirements of the user's manual, two knowledge bases, called "pets.kb" and "wine.kb" (see Appendices F and G), were developed from the instructions given in the user's manual. These two knowledge bases were designed in advance to verify ABC's ability to integrate OAV triples with frames and to deal with uncertainties. The successful execution of the "pets.kb" knowledge base also demonstrated ABC's capability of frame inheritance and demon procedures.

The user-interface was never tested in any quantitative manner, but it was presented to the sponsor for approval under the rapid prototyping methodology as explained earlier in Chapter I.

IX. Conclusions and Recommendations

This thesis succeeded in developing an expert system shell which integrated OAV triples into frames. Not only did it provide a working expert system shell, but more importantly, it provided insight into making several significant enhancements. These enhancements, if implemented, would increase the capabilities of the expert system shell developed in this thesis and as a side effect, make it more efficient and user friendly.

This chapter will briefly summarize the thesis and then provide an assessment of the ABC expert system shell. It will then cover several relevant aspects of the design and implementation process which, for better or worse, shaped this thesis. The chapter will conclude by discussing recommendations for future improvements.

Summary

This thesis investigated how to develop an expert system shell which integrated object-attribute-value (OAV) triples with frames and implemented the shell entirely in standard Prolog. Rather than starting from scratch, the approach used in this thesis was to study and expand upon an existing educational expert system shell called BC3. BC3 was chosen because it was an educational expert system shell which symbolized its knowledge in OAV triples. Once the decision to expand upon BC3 was made, the thesis was divided into four phases.

In phase one, the functions of BC3 were studied to determine BC3's strengths and weaknesses. This phase also provided a period of time to get better acquainted with Prolog. It was during this phase that the reasoning mechanism in BC3 was altered to make it more efficient. Additionally, the "why" and "how" trace mechanisms of BC3 were modified to provide measurable improvements.

Phase two investigated different approaches to creating a user interface for an expert system shell. In this phase, two predicates were created which eliminated the requirement to enter data in the form of a Prolog structure. More importantly, a command line scheme similar to Teknowledge's M.1 expert system shell was implemented to modularize the top level of ABC into functional divisions. This phase was also where several predicates were created to make a knowledge base more "English-like" in order for it to be easier to create and maintain. Additionally, it was during this phase that the prompts generated from ABC askables were customized and the valid replies to an askable were enumerated. These additions not only enhance the look of ABC but also make the questions more understandable and the answering process more studentproof.

Phase three investigated how to integrate frames, including demon procedures, into an expert system shell symbolizing its knowledge with OAV triples. The method implemented in ABC uses the frame's name to represent an object, the slot's name to represent an attribute, and the slot's value to represent a list of all values which the object-attribute pair can take on. Additionally, predicates to do the fundamental frame-base maintenance were created.

Phase four investigated inexact reasoning techniques and integrated the capability of inexact reasoning into ABC with the use of certainty factors. The inferencing mechanism had to be altered during this phase from previous phases to permit multiple solutions.

Assessment

Several conclusions were drawn from this thesis effort. The first conclusion was that Prolog makes a very good rapid prototyping tool. It lends itself to both bottom-up or top-down development fairly easy. Prolog is weak in the area of Input/Output (I/O) but was otherwise ideally suited for this thesis effort.

Secondly, I discovered some pitfalls in using the rapid prototyping methodology. Step two of the rapid prototyping methodology requires the task to be divided into subtasks. The creation of subtasks in developing ABC was straightforward and natural. However, the order in which these subtasks were undertaken -- reviewing BC3, developing a user-interface, integrating frames, and integrating certainty factors -- was a major mistake and made the rapid prototyping methodology appear inappropriate during the development of the last subtask.

The integration of OAV triples and a simple frame-base representation was very natural. This was expected to be the hardest of the four subtasks before the thesis had begun but was in retrospect, the easiest of the subtasks to complete. In contrast, implementing certainty factors was initially thought to be the simplest task and was definitely the most difficult to implement. The implementation of certainty factors required ABC to do considerable amounts of

backtracking. Until the implementation of certainty factors, ABC was written to perform very efficiently without backtracking. Thus the last subtask, implementing certainty factors, required a major revision of the code written in the previous two phases of implementation.

Recommendations

All of the recommendations can fit into two main categories: quick fixes and long-term enhancements. Most of the quick fixes are items which the author intended to place into ABC but, because of time constraints, could not implement. The long-term enhancements are a result of knowledge learned during the span of this thesis effort. These enhancements can provide significant flexibility and power to ABC and should be implemented as soon as possible.

Quick Fixes. Currently, ABC will continue searching for solutions until it finds a solution with a certainty factor of 100 or until its search paths are exhausted. This is fairly inefficient and can be corrected by a combination of two features. The first of the two features needed is a 'multivalued' predicate similar to the one found in M.1. The 'multivalued' predicate should prevent search for additional solutions to a goal unless the goal is specifically declared to have multiple values. The second feature needed is to prevent ABC from searching for additional solutions if it finds a solution with a certainty factor slightly less than 100, possibly 90 or 95. Implementation of these two features would greatly enhance ABC's efficiency when dealing with uncertainties.

Even though all of the tests which were run against ABC did so flawlessly, it is recommended that ABC be tested with several students from the introductory AI course. This type of testing would assist in getting any ambiguities out of the user's manual along with removing any remaining bugs in the program itself.

Long-Term Enhancements. ABC stores facts, frames, rules, and askables in the Prolog database using different structures. In addition, OAV triples can be confirmed or denied. Having all of these structures is inefficient. There are only three structures needed: one for OAV triples which can be asserted (facts, frames, confirmed triples, etc.), a second structure for rules, and a third structure for an askable. Furthermore, if the first structure for assertables were structured as shown in Figure 9-1, there would be several major side benefits.

```
assertable( Obj, Attr, Val, CF, [ * ], [ ** ], [ *** ])
```

* is a list indicating the reason for the triple's assertion along with the certainty factor associated with that reason.

** is a list of facets or attributes

*** is an explanation of the OAV triple to be used with the explanation facility.

Figure 9-1: Recommended Structure of ABC Assertable

With the implementation of the structure change recommended in the above paragraph and shown in Figure 9-1, the trace mechanism can be

greatly simplified and enhanced. The trace mechanism can use the reason for a triple's assertion, the first list, as a pointer, pointing to the next assertion which needs explaining. Both M.1 and GoldWorks use a pointer technique in their explanation facility. Additionally, if explanatory text is provided, it can be presented to the end-user adding another dimension of user-friendliness to the expert system shell.

Each frame, when read by the expert system shell, can be decomposed with each slot and associated slot-values and slot-attributes being asserted as an ABC assertable (See Figure 9-1). This is similar to how GoldWorks handles its frames (13:75).

This structure would support such future enhancements as retracting assertables whenever their reason for being asserted is retracted. This feature would be invaluable if a shell for nonmonotonic reasoning were needed.

The next long-term enhancement would be to include additional on-line help and edit facilities to both the end-user and the expert system developer. The developer should be able to edit any part of the knowledge base from the ABC prompt.

The last long-term enhancement would be to use the Arity Prolog compiler and compile ABC. This enhancement would have several immediate benefits. Using Arity's superset of standard Prolog, 30 to 40 percent of the source code could be deleted. Additionally, Arity Prolog supports a string type which is much more efficient than using standard Prolog lists. Windows could quickly be added to further enhance the user's interface. Lastly, the code will be in an executable form with AFIT as the sole owner of the data rights. This would allow ABC to be

distributed freely, to run on any number of machines available at AFIT
and throughout the Air Force.

Bibliography

1. Amsterdam, Jonathan. "Building a Flexible Knowledge Representation Scheme" AI Expert, Vol. 1, No. 11: 19-22 (November 1986).
2. Barr, A. et al. The Handbook of Artificial Intelligence, Volume 1. Los Altos CA: William Kaufman Publishing Co., 1982.
3. Bratko, Ivan. Prolog Programming For Artificial Intelligence. Wokingham England: Addison-Wesley Publishing Co., 1986.
4. Buchanan, Bruce G. and Edward H. Shortliffe. Rule-Based Expert Systems. Reading MA: Addison-Wesley Publishing Co., 1984.
5. Clocksin, William F. and C. S. Mellish. Programming in Prolog. Berlin: Springer-Verlag, 1981.
6. Cuadrado, J.L. and C.Y. Cuadrado. "AI in Computer Vision," BYTE, Vol. 11, No. 1: 237-258 (January 1986).
7. Fikes, Richard and Tom Kehler. "The Role of Frame-Based Representation in Reasoning," Communications of the ACM, Vol. 28, No. 9: 904-920 (September 1985).
8. Frenzel, Louis E., Jr.,. Understanding Expert Systems. Indianapolis IN: Howard W. Sams and Co., 1987.
9. GoldWorks. Expert System User's Guide Version 1.0. Gold Hill Computers, Inc., Cambridge, MA, April 1987.
10. Hardy, Steve. "Re : Knowledge Representations and Prolog". Electronic Message. 23253@teknowledge-vaxc.ARPA, 17 Jun 88.
11. Hu, David. Programmer's Reference Guide to Expert Systems. Indianapolis IN: Howard W. Sams and Co., 1987.
12. KEE. Interface Reference Manual K3.1-IRM-1. Intellicorp Inc., USA, 1987.
13. Levine, Ken. "The Age of GoldWorks," PC TECH Journal, Vol. 6, No. 5: 68-81 (May 1988).
14. M.I. Training Materials Manual M1-020-3001-04. Teknowledge Inc., Palo Alto, CA, 1984.
15. Marcus, Claudia. Prolog Programming. Reading MA: Addison-Wesley Publishing Co., 1986.
16. Mishkoff, Henry C. Understanding Artificial Intelligence. Indianapolis IN: Howard W. Sams and Co., 1985.

17. Partridge, D. Artificial Intelligence Applications in the Future of Software Engineering. Chichester England: Ellis Horwood Ltd., 1986.
18. Personal Consultant Plus. User's Manual 2539262-0001. Texas Instruments Inc., Austin, TX, 1986.
19. Pressman, Roger S. Software Engineering. New York: McGraw-Hill Book Co., 1987.
20. Tanimoto, Steven L. The Elements of Artificial Intelligence. Rockville MD: Computer Science Press, Inc., 1987.
21. Walker, Adrian et al. Knowledge Systems and Prolog. Reading MA: Addison-Wesley Publishing Co., 1987.
22. Waterman, Donald A. A Guide to Expert Systems. Reading MA: Addison-Wesley Publishing Co., 1986.

APPENDIX A

The Original BC3.PRO Expert System Shell

This appendix contains the expert system shell BC3 as it was originally written before this thesis effort. It is this shell in which ABC takes its roots. BC3 has several good features: the ability to explain its conclusion, the ability to inquire about a prompt which it makes, and the ability to mix Prolog code within BC3 code when appropriate. However, BC3 does have some drawbacks. It does not have a frame representation language or the inferencing mechanism to extract knowledge from frames. It doesn't have the capability to deal with incomplete or uncertain knowledge. It can only deal with production rules which look like the following example.

```
rule_1:  if    [Whoever, saves_their, money]    and
           [Whoever, studies, hard]             or
           [Whoever, has_a, rich_daddy]
        then [Whoever, can_go_to, college].
```

It also has the facilities to store facts and "askables." Askables are Object-Attribute-Value (OAV) triples which have to be completed or confirmed by prompting the user for input. Askables are similar in nature to meta-rules in M.1.

```

/*****
/*
/*          BC3
/*
/*      A shell for backward-chaining expert systems.
/*
/*      Each item of knowledge is represented by a triple (i.e.,
/*      a three-element list of the form [Object,Attribute,Value].
/*      An associated rule-base supplies the following data:
/*
/*      1. A goals-statement, in the form of a list of triples to
/*          be solved in sequence. The solved triples are printed
/*          by the shell.
/*      2. A collection of if-then rules for triples.
/*      3. A collection of facts, i.e., triples asserted as known
/*          a priori.
/*      4. A collection of 'askable' triples, indicating the forms
/*          of triples whose values may be obtained from the user.
/*      5. A collection of 'keep' triples, indicating the form of
/*          the triples not to be erased from working memory at
/*          the beginning of a new session.
/*
/*      Each item of knowledge stored in working memory is of the
/*      form confirmed([Obj,Attr,Val]) or denied([Obj,Attr,Val]).
/*
/*      To use the system, load BC3, load the appropriate rule-
/*      base and type 'start.' Because BC3's operator-defini-
/*      tions are used by the rule-bases, BC3 must load first.
/*
*****/

```

```

/*----- OPERATOR DEFINITIONS -----*/
/*
/*      The operators defined below enable the rules in the know-
/*      ledge-base to be expressed in a form more readable than
/*      the standard (prefix) form.
/*
/*-----*/

```

```

?- op(250, xfx, :).
?- op(245, xfx, then).
?- op(240, fx, if).
?- op(235, xfx, derived_from).
?- op(230, xfy, or).
?- op(225, xfy, and).

```

```

/*----- START -----*/
/*
/* The procedure 'start' begins by erasing from working mem- */
/* ory all 'confirmed' and 'denied' clauses, except those */
/* clauses protected by 'keep' from erasure. The list of */
/* goal-triples is then read from the rule-base and solved in */
/* turn by 'solve'. A trace is maintained of the back- */
/* chaining search-tree generated in solving the goals. When */
/* the last of the goal-triples is solved, the values of all */
/* goals, except those solved by asking the user directly, */
/* are displayed; the trace is also displayed, if requested, */
/* as a "how" explanation of the solution. */
/*
/*-----*/

start :-
    ask_about_loading_wm,          /* Erase all working-mem- */
    ( confirmed(Triple),          /* ory elements not pro- */
      not(keep:Triple),          /* tected by 'keep' state-*/
      retract(confirmed(Triple)) /* ments in the knowledge-*/
    ;                             /* base. */
      denied(Triple),
      not(keep:Triple),
      retract(denied(Triple)) ),
    fail.

start :-
    retract_all(why_trace(_)),    /* Erase the "why" trace. */
    goals: Goals,                /* Find the goal-triples, */
    prefix(Goals,Prefixed_goals), /* prefix each of them */
    reverse(Prefixed_goals,Goal_list), /* with the word 'goal', */
                                     /* & reverse their order. */
    solve(Goals,[],Part_trace),   /* Satisfy all of the */
    !,nl,                        /* goals and then put the */
    append(Goal_list,Part_trace,Trace),
                                     /* list of goals at the */
                                     /* front of the "how" */
    ask_about_trace(Trace),        /* trace. Supply a "how" */
    ask_about_saving_wm.          /* explanation on request.*/

start :-                          /* If all triples can't */
    nl,                          /* be solved, announce it.*/
    write('I can''t solve this problem. '),nl.

```

```

/*----- SOLVE -----*/
/*
/* The predicate 'solve(Goals,Trace,New_trace)' means that
/* Goals is a list of goals (expressed as triples), and that
/* Trace and New_trace are, respectively, the trace-lists be-
/* and after solution of the goal at the head of the goal-
/* list. The procedure 'solve' solves each of the goals in
/* turn. The first step in solving a goal is to erase the
/* "why" trace and to initialize it with that goal. Thus each
/* goal is solved with a separate "why" trace. As each rule
/* is encountered in descending through the search-tree for a
/* given goal, that rule is added to the front of the "why"
/* trace.
/*
/*-----*/

```

```

solve([],Trace,Trace).
solve([Goal|Others],Trace,New_trace) :-
    retract_all(why_trace( )),          /* Initialize the "why" */
    asserta(why_trace([goal:Goal])),    /* trace. */
    is_known(Goal,Trace,Trace1),
    ( confirmed(Goal),!                  /* Write each triple as */
    ;                                   /* it's solved, but don't */
      nl,write_triple(Goal),nl ),        /* write a triple that's */
    solve(Others,Trace1,New_trace).      /* been told explicitly */
                                         /* by the user. */

```

```

write_triple([Obj,Attr,Val]) :-
    writelist([Obj,' ',Attr,' ',Val,'.']).

```

```

ask_about_trace(Trace) :-
    write('Do you wish to see how this answer was arrived at? '),
    read(Reply),
    ( means(Reply,yes), !,
      write_trace(Trace)
    ;
      true ).

```

```

ask_about_loading_vm :-
    write('Do you wish to load from a working-memory file? '),
    read(Reply),nl,
    ( Reply = y,
      load_working_memory, !
    ;
      true ).

```



```

ask_about_saving_wm :-
    write('Do you wish to save working memory in a file? '),
    read(Reply),nl,
    ( Reply = y,
      save_working_memory, !
    ;
      true).

prefix([],[]).
prefix([Goal|Goals],[goal:Goal|Prefixed_Goals]) :-
    prefix(Goals,Prefixed_Goals).

/*----- IS_KNOWN -----*/
/*
/* The 'is known' procedure maintains a trace of the path of
/* the solution-tree leading to the triple currently under
/* consideration. 'is known(Triple,Trace,New_trace)' means
/* that if reasoning to a certain point has been recorded in
/* the list 'Trace', then the additional triple 'Triple' is
/* known via reasoning recorded by the list 'New_trace'.
/*
/*-----*/

/* A triple is not known if it has been denied by the user. */

is_known(Triple,Trace,Trace) :-
    denied(Triple),
    !,
    fail.

/* An O-A-V triple is known if it is a fact in the rule base. */

is_known([O,A,V],Trace,Trace) :-
    member(fact:[O,A,V],Trace),
    !.

is_known([O,A,V],Trace,[fact:[O,A,V]|Trace]) :-
    fact: [O,A,V],
    !.

/* A triple is known if it has been confirmed by the user. */

is_known(Triple,Trace,Trace) :-
    member(was_told:Triple,Trace),
    !.

is_known(Triple,Trace,[was_told:Triple|Trace]) :-
    confirmed(Triple),
    !.

```

```

/* A triple [X,P,Y] is known if the Prolog goal P(X,Y) suc- */
/* ceeds, either because P is a built-in predicate, or because */
/* the rule-base has prolog-code defining P. The triple */
/* [2,member,[1,2]], for example, is converted into the goal */
/* member(X,[1,2]), which is then executed by Prolog. To keep */
/* non-Prolog-programmers out of trouble, the triple [X,is,Y] */
/* is trapped so that it will not be executed as an arithmetic */
/* statement. The triple [X,=,Y] is interpreted as the Prolog */
/* goal X is Y. */

```

```

is_known([Obj,Attr,Val],Trace,Trace) :-
    member(solved:[Obj,Attr,Val],Trace),
    !.

```

```

is_known([Obj,Attr,Val],Trace,[solved:[Obj,Attr,Val]|Trace]) :-
    not (Attr == is),          /* We don't want 'is' to be inter- */
    T =.. [Attr,Obj,Val],     /* preted arithmetically. */
    T,
    !.

```

```

is_known([Obj,=,Val],Trace,[solved:[Obj,=,Val]|Trace]) :-
    Obj is Val.               /* Interpret '=' as Prolog's 'is' */
                               /* predicate. */

```

```

/* A triple is known if it is the head of a rule and the con- */
/* ditions of the rule are satisfied. We put a rule that we */
/* encounter at the head of the "why" trace, erasing any du- */
/* plicates of the rule that are already in the "why" trace. */
/* The "why" trace is maintained in the database, in a clause */
/* of the form 'why_trace(<List of goals and rules>)'. This */
/* differs from the "how" trace, which is handed as an argu- */
/* ment from goal to goal. */

```

```

is_known(Triple,Trace,[was_proved:[Triple,Rule]|Trace]) :-
    member(Rule: Triple derived_from Conds,Trace),
    !.

```

```

is_known(Trpl,Trc,[Rule: Trpl derived_from Conds|Trcl]) :-
    Rule: if Conds then Trpl,
    /*

```

```

        TAKE OUT CODE ACCESSING THE WHY-EXPLANATION
        why_trace(Why_trace),
        remove(Rule: Trpl derived_from Conds,Why_trace,Part_why),
        append([Rule: Trpl derived_from Conds],Part_why,New_why),
        retract(why_trace( )),
        asserta(why_trace(New_why)),
    */
    is_known(Conds,Trc,Trcl),
    !.

```

```

/* A condition involving "and", "or", or "not" is known if its */
/* parts are known in suitable combinations. */

```

```

is_known(Triples1 and Triples2,Trace,Trace2) :-
    is_known(Triples1,Trace,Trace1),
    is_known(Triples2,Trace1,Trace2).

is_known(Triples1 or Triples2,Trace,Trace1) :-
    is_known(Triples1,Trace,Trace1).
is_known(Triples1 or Triples2,Trace,Trace2) :-
    is_known(Triples2,Trace,Trace2).

is_known(not Triple,Trace,[confirmed_not:Triple|Trace]) :-
    not is_known(Triple,Trace,Trace1).

/* A triple is known if (a) the rule-base classifies it as      */
/* "askable" and if (b) the user confirms it. The user may      */
/* request a "why" explanation before responding to the ques-   */
/* tion.                                                          */

is_known([0,A,V],Trace,Trace) :-
    member(was_told:[0,A,V],Trace),
    !.

is_known([0,A,V],Trace,[was_told:[0,A,V]|Trace]) :-
    askable: [0,A,_],
    ask_about([0,A,V]),
    !,
    confirmed([0,A,V]).

/*----- ASK ABOUT -----*/

ask_about([Obj,Attr,Val]) :-
    var(Val),
    !, nl,
    writelist([Obj,' ',Attr,'? ']),nl,
    askable: [Obj,Attr,Legal_values],
    write('Legal values: '), write(Legal_values), nl,
    write('> '), read(Reply),
    ( (
        means(Reply,why),          /* If the user responds */
        explain_why([Obj,Attr,Val]), /* with 'why.', give him */
        !,                          /* an explanation.      */
        ask_about([Obj,Attr,Val])
    );
    (
        atomic(Legal_values)
    ;
        member(Reply,Legal_values)
    ), !,
    assertz(confirmed([Obj,Attr,Reply]))
;
    write('Please re-enter your reply. '),nl,
    ask_about([Obj,Attr,Val])
).

```

```

ask_about([Obj,Attr,Val]) :-
    nl,
    writelist([Obj,' ',Attr,' ',Val,'? (yes./no./why.)']),
    nl,write('> '),read(Reply),
    (
        means(Reply,yes),
        assertz(confirmed([Obj,Attr,Val])), !
    ;
        means(Reply,no),
        assertz(denied([Obj,Attr,Val])), !
    ;
        means(Reply,why),
        explain_why([Obj,Attr,Val]),
        !,
        ask_about([Obj,Attr,Val])
    ;
        write('Please re-enter your reply. '),nl,
        ask_about([Obj,Attr,Val])
    ).

means(Reply,yes) :-
    member(Reply,[y,yes]).
means(Reply,no) :-
    member(Reply,[n,no]).
means(Reply,why) :-
    member(Reply,[why,w]).

/*----- EXPLAIN WHY -----*/

explain_why(Triple) :-
    why_trace(Why_trace),
    write('Because: '),nl,
    justify(Triple,Why_trace).

justify(Triple,Why_trace) :-
    member(goal:Goal,Why_trace),
    Triple = Goal,
    writelist(['This will satisfy the goal ',Goal]),nl,
    nl,
    !.

justify(Triple,Why_trace) :-
    member(R:Head derived_from Cs,Why_trace),
    among(Triple,Cs),
    remove(R:Head derived_from Cs,Why_trace,New_trace),
    writelist(['I can use ',Triple]),nl,
    list_known_triplets(Cs),
    writelist(['      to help satisfy ',R,': ',Head]),nl,nl,
    justify(Head,New_trace).

```

```

list_known_triples(Cs) :-
    among(Triple,Cs),
    (
        confirmed(Triple)
        ;
        fact: Triple
    ),
    writelist(['    knowing ',Triple]),nl,
    fail.
list_known_triples(_).

among(Triple,Conditions) :-
    Triple = Conditions.
among(Triple, First_triple and Other_conditions) :-
    Triple = First_triple,!
    ;
    among(Triple,Other_conditions).
among(Triple, First_triple or Other_conditions) :-
    Triple = First_triple,!
    ;
    among(Triple,Other_conditions).

/*----- WRITE_TRACE -----*/

write_trace([]) :-
    nl.
write_trace([goal:Triple|Rest]) :-
    !,
    write('GOAL:  '),write(Triple),nl,
    write_trace(Rest).
write_trace([fact:Triple|Rest]) :-
    !,
    write('FACT:   '),write(Triple),nl,
    write_trace(Rest).
write_trace([solved:Triple|Rest]) :-
    !,
    write('SOLVED: '),write(Triple),nl,
    write_trace(Rest).
write_trace([was_told:Triple|Rest]) :-
    !,
    write('TOLD:    '),write(Triple),nl,
    write_trace(Rest).
write_trace([confirmed_not:Triple|Rest]) :-
    !,
    write('CONTRADICTED: '),write(Triple),nl,
    write_trace(Rest).
write_trace([was_proved:[Triple,Rule]|Rest]) :-
    !,
    write('PROVED: '),write(Triple),write(' USING '),write(Rule),nl,
    write_trace(Rest).
write_trace([Rule: Triple derived_from Conditions|Rest]) :-

```

```

!,
writelst([Rule,': ',Triple,' Was Derived From']),nl,
write_conditions(Conditions),
write_trace(Rest).
write_trace([X|Rest]) :-
    write(X),nl,
    write_trace(Rest).

write_conditions([X,Y,Z]) :-
    tab(8),write([X,Y,Z]),nl.
write_conditions(not [X,Y,Z]) :-
    tab(4),write('NOT '),write([X,Y,Z]),nl.
write_conditions([X,Y,Z] and Conditions) :-
    tab(8),write([X,Y,Z]),write(' AND'),nl,
    write_conditions(Conditions).
write_conditions(not [X,Y,Z] and Conditions) :-
    tab(4),write('NOT '),write([X,Y,Z]),write(' AND'),nl,
    write_conditions(Conditions).
write_conditions([X,Y,Z] or Conditions) :-
    !,
    tab(8),write([X,Y,Z]),write(' OR'),nl,
    write_conditions(Conditions).
write_conditions(Conditions1 or Conditions2) :-
    write_conditions(Conditions1),tab(8),write('OR'),nl,
    write_conditions(Conditions2).
write_conditions(not [X,Y,Z] or Conditions) :-
    tab(4),write('NOT '),write([X,Y,Z]),write(' OR'),nl,
    write_conditions(Conditions).

/*----- FILE I/O -----*/

save_working_memory :-
    write('Please supply a filename: '),
    read(Filename),nl,
    tell(Filename),
    save_wme,
    told.

save_wme :-
    confirmed(Triple),
    writeq(confirmed(Triple)),write('.'),nl,
    fail.
save_wme :-
    denied(Triple),
    writeq(denied(Triple)),write('.'),nl,
    fail.
save_wme.

```

```

load_working_memory :-
    write('Please supply a filename: '),
    read(Filename),nl,
    retract_all(confirmed(_)),
    retract_all(denied(_)),
    see(Filename),
    loadfile,
    write('Contents of working memory:'),nl,nl,
    list_working_memory,
    seen.

loadfile :-
    read(Term),
    load(Term).

load(end_of_file) :-
    !.
load(Term) :-
    not Term =.. [confirmed,_],
    not Term =.. [denied,_],
    !,
    write('Not a legal file of working-memory elements...'),nl,nl,
    retract_all(confirmed(_)),
    retract_all(denied(_)).
load(Term) :-
    assertz(Term),
    loadfile.

list_working_memory :-
    confirmed(Triple),
    write(confirmed(Triple)),write(' '),nl,
    fail.
list_working_memory :-
    denied(Triple),
    write(denied(Triple)),write(' '),nl,
    fail.
list_working_memory.

/*----- UTILITY PROCEDURES -----*/

writelist([]).
writelist([X|L]) :-
    write(X),
    writelist(L).

member(X,[X|_]).
member(X,[_|L]) :-
    member(X,L).

```

```

append([],L,L).
append([X|L],M,[X|N]) :-
    append(L,M,N).

reverse([],[]).
reverse([X|L],M) :-
    reverse(L,N),
    append(N,[X],M).

remove(_,[],[]).
remove(X,[X|L],M) :-
    !,
    remove(X,L,M).
remove(X,[Y|L],[Y|M]) :-
    remove(X,L,M).

retract_all(X) :-
    not not retract(X),
    retract_all(X).
retract_all(X) :-
    not not retract((X :- Y)),
    retract_all(X).
retract_all(_).

wm :-
    listing(confirmed),
    listing(denied).

reset :-
    retract_all(confirmed(_)),
    retract_all(denied(_)).

again :-
    write('Consulting bc3.pro'),nl,
    reconsult('bc3.pro').

why :-
    why_trace(Trace),
    write_trace(Trace).

?- write('Type ''start.'' to begin. '),nl,nl,
    write('Answer all questions using lower case, '),nl,
    write('ending with a period. '),nl.

```



```

/*****
/*
/*          ABC - AFIT Backward Chainer Version One          */
/*                      04 Oct 88                      */
/*
/* ABC is an expert system shell written entirely in Clocksin and */
/* Mellish Prolog. It reasons with a backward inference engine and */
/* a forward reasoning mechanism called "initial_askable." It uses */
/* production rules and frames for its knowledge representation. */
/* The production rules are made up of Object-Attribute-Value (OAV) */
/* triples representing relationships that an object may have. OAV */
/* triples were used in MYCIN. The frame structure is simple but */
/* effective for smaller knowledge bases. The frames allow hier- */
/* archical inheritance of values. Also, the frame language */
/* provides the use of demons for slots which are designated as */
/* "if needed, if added, or if removed." Additionally, ABC has the */
/* ability to deal with uncertain knowledge via its use of certainty */
/* factors, CFs. */
/*
/*-----
/*----- Operator Definitions -----*/

:- op(990, xfx, =).
:- op(980, xfy, :).
:- op(975, xfx, then).
:- op(970, xfy, slot).
:- op(970, fx, if).
:- op(965, xfx, derived_from).
:- op(960, xfy, or).
:- op(955, xfy, and).
:- op(950, xfx, cf).

/*----- Start Predicate -----*/
/*
/* The start predicate "initiates" the database and keeps subsequent */
/* operations in an infinite loop via the repeat/0 and execute/1 */
/* predicates. This loop prompts the user for an ABC command. Each */
/* of these commands will ultimately fail (except "quit") causing */
/* backtracking to the repeat predicate where the cycle repeats with */
/* another prompt. There are dozens of ABC commands which, relative */
/* to the start predicate, work on the principle of "side-effects." */
/* The start predicate is called by the Prolog interpreter upon */
/* consulting the ABC shell. You may exit the ABC shell by typing */
/* "quit." at the ABC prompt. */
/*
/* Constraints: None

```

```

start :-                                /* ABC initiates itself by clearing */
    cls,                                /* the screen and displaying its intro */
    display_intro_screen,              /* screen. It also restores the know- */
    restore_kb,                        /* ledge base. It then goes into its */
    nl,nl,                             /* loop of executing an ABC command, */
    repeat,                            /* failing, repeating its prompt, */
    nl,nl,                             /* executing the next command, etc */
    write('ABC > '),                   /* until the ABC quit command is seen. */
    readline([],Reply),
    execute(Reply).

/*----- display_intro_screen -----*/
/*
/* The display_intro_screen predicate simply displays the intro-
/* ductory message on the CRT.
/*
/* Constraints: None
*/

display_intro_screen :-
    put(7),
    nl,nl,nl,
    tab(27),
    write('ABC - AFIT Backward Chainer'),
    nl, tab(28),
    write('Version 1.0 - 27 Sep 88'),
    nl,nl,nl,
    rules.

/*----- rules Predicate -----*/
/*
/* The rules predicate displays the syntactical rules which must be
/* obeyed throughout any consultation with ABC in order for it
/* to operate correctly. The ABC command "rules" also calls this
/* this predicate so the user may view the syntax rules at any ABC
/* prompt.
/*
/* Constraints: None
*/

rules :-
    nl,nl,tab(32),
    write('ABC Syntax Rules'),
    nl,tab(14),
    write('====='),
    nl,nl,tab(14),
    write('1. Always use lower case letters.'),
    nl,nl,tab(14),
    write('2. Always enter in single words at command prompts.'),
    nl,tab(14),
    write('   (i.e. review_frame is a single word)'),
    nl,nl,tab(14),
    write('3. Always end your command with a <Return>.'),
    nl.

```

```

/*----- help -----*/
/*
/* The ABC command called HELP is a single level screen display which */
/* will aid the user as a memory assistant. Its function may be      */
/* altered to fit the needs of the intended user. The help command is */
/* just one of many Prolog rules which make up the "execute" predi-   */
/* cate. The user must provide a valid ABC command at the ABC prompt  */
/* in order for the "execute" predicate to operate. If a non-valid    */
/* command is provided, a beep followed by an error message will     */
/* result.                                                             */
/*                                                                     */
/* Constraints: execute(+) The command must be instantiated.          */
/*-----*/

execute(help) :-
    cls,
    nl,nl,
    tab(29),
    write('ABC Online Help Display'),
    nl, tab(10),
    write('====='),
    nl,nl, tab(10),
    write('load <kb> . . . . Load a parsed/unparsed knowledge base'),
    nl,nl, tab(10),
    write('remove kb . . . . Removes entire kb from Prolog database.'),
    nl,nl, tab(10),
    write('consultation (go) Tries to find a solution to the goal'),
    nl,nl, tab(10),
    write('restore . . . . Restores working memory for new consultation'),
    nl,nl, tab(10),
    write('quit . . . . . Leave the ABC shell for the Prolog
interpreter'),
    nl,
    !,
    fail.

/*----- quit -----*/
/*
/* The ABC quit command is the only way to exit ABC in a normal way. */
/*-----*/

execute(quit).

/*----- load -----*/
/*
/* The load command will load in either a parsed ".abc" file or an   */
/* unparsed ".kb" file. The ".abc" files are read directly into the  */
/* memory. However, the ".kb" files have to be parsed and their      */
/* quoted triples converted to triples within lists. Additionally,   */
/* the user is given the opportunity to load in a working memory file*/
/* which should contain all the frames, facts, and information       */
/* learned from previous consultations. Last, the user is given the  */
/* option of running a new consultation directly without going back   */
/* to the ABC Executive.                                             */
/*-----*/

```

```

/* Constraints: Filenames provided by the user must exist. System */
/* error warnings will be produced otherwise. */

execute(load) :-
    cls,
    nl,nl,
    write('Enter the name of the file where your knowledge base is stored,
or'),
    nl, write('enter <Return> to abort.'),
    nl,nl,
    write('Filename, including path is: '),
    readline([],Filename),nl,
    ( Filename = '',!,fail          /* If user just pressed <ENTER>, then */
    ;                               /* abort the load operation. */
    true
    ),
    nl,nl,nl,nl,                  /* If the filename has the extension */
    ( check(Filename,abc),        /* ".abc" then just load in the file */
    write('Reading the knowledge base called '), /* using the pred- */
    write(Filename),              /* cate called "load_abc /1". */
    write('.') ,
    nl,nl,
    write('Please standby.'),
    load_abc(Filename)
    ;
    check(Filename,kb),          /* If the filename has the extension */
    write('Reading and Parsing the knowledge base called '), /* the */
    write(Filename),             /* knowledge base, generate the ".abc"*/
    write('.') ,                /* file, and then read in the ".abc" */
    nl,nl,                      /* file. This is accomplished by */
    write('Please standby.'),    /* using the "load_kb /1" predicate. */
    load_kb(Filename)
    ;
    write('File extension is incorrect. Load procedure aborted.'),
    !, fail
    ),
    nl,nl,nl,                    /* If the file has the wrong extension*/
    write('Knowledge base '),    /* then write an error message and */
    write(Filename),            /* abort the load procedure. */
    write(' has been successfully read.'),
    nl,nl,
    write('Would you like to load in a working memory file? (<y>,n) '),
    get_reply(Answer),
    ( Answer = yes,              /* See if user wants to load working */
    load_wm(Filename)          /* memory. If reply is yes, then load */
    ;                           /* in working memory using the "load_ */
    true                       /* wm" predicate, otherwise continue. */
    ),
    nl,nl,

```

```

write('Would you like to load in an auxiliary file? (<y>,n) '),
get_reply(Answr),
( Answr = yes,          /* Likewise, see if the user wants to */
  load_aux(Filename)   /* load in an auxiliary file. If reply*/
;                      /* is yes, load in the auxiliary file */
  true                 /* using the "load_aux /1" predicate. */
),
nl,nl,nl,
write('Would you like to start a consultation now? '),
get_reply(Reply),
( Reply = yes,          /* Next see if the user wants to start*/
  execute(consultation) /* a consultation with the knowledge */
;                      /* base, assuming one is loaded. If */
  true                 /* reply is yes, then start a consul- */
),                      /* tation using the ABC command "con- */
!,                      /* sultation". Either way, return to */
fail.                  /* the ABC prompt when finished. */

/*----- load_kb -----*/
/*
/* load_kb/1 loads in an unparsed knowledge base, parses it, writes
/* it to a file with the same prefix name, yet with a file extension
/* of ".abc". This predicate will then read from the new ".abc"
/* file, inserting terms into working memory as it reads them.
/*
/* Constraints: load_kb(+)
/* The entire file name must be passed to load_kb, including the
/* path. Since this path/file name must be an atom, the path/file
/* name must be surrounded by single quotes for proper operation.
/* Additionally, if the filename does not exist, the system will
/* produce an error message which may be impossible to recover from
/* without rebooting the system and loosing your data.
*/

load_kb(Filename) :-      /* The FULL filename (w/extension) */
  check(Filename,kb),     /* must be entered. If the extension */
  !,                      /* is ".kb", then we read the file */
  see(Filename),           /* a term at a time, and */
  convert_filename(Filename,abc,New_FN), /* transform it to ".abc" */
  tell(New_FN),           /* format and write it out */
  makefile,               /* to a file with the same name but */
  seen,                   /* with an extension of ".abc". */
  told,                   /* See the comments for convert_file-*/
  see(New_FN),            /* name, check, makefile, and */
  loadfile,               /* loadfile for additional details. */
  seen.

```

```

/*----- load_abc -----*/
/*
/* The load_abc predicate simply reads in terms from the file which
/* the user provides the name for, and asserts the terms in the
/* Prolog database.
/*
/* Constraints: The same constraints as load_kb apply.
*/

```

```

load_abc(Filename) :-
    see(Filename),
    loadfile,
    seen.

```

```

/*----- load_wm -----*/
/*
/* The load_wm predicate takes the ".abc" or ".kb" knowledge base
/* filename, creates a filename with the same prefix yet with an
/* extension of ".wm" and then searches the current drive for the
/* file with that name. The file is then read in much like the ".kb"
/* files, parsed, read back to a temporary file called "abc.tmp" and
/* then read in using the Prolog reader and asserted into the Prolog
/* database. This is necessary to convert quoted triples to listed
/* triples and to get the Prolog database to accept variables as
/* variables instead of quoted atoms.
/*
/* Constraints: Basically the same as the "load_kb" predicate. The
/* system will produce an error if the working memory file does not
/* exist.
*/

```

```

load_wm(KB_Filename) :-
    !,
    convert_filename(KB_Filename, wm, WM_Filename),
    see(WM_Filename),
    tell('ABC.TMP'),
    makefile,
    seen,
    told,
    retract_all(frame : Framename slot Slots),
    retract_all(confirmed(Triples)),
    retract_all(denied(Triples)), /* The old working memory is de-
    see('ABC.TMP'),              /* stroyed when loading in a new
    loadfile,                     /* working memory.
    seen, !.
*/

```

```

/*----- load_aux -----*/
/*
/* The load_aux predicate is used to load user-defined Prolog pred- */
/* icates into the Prolog database. This includes the expert system */
/* developer's demons, if they exist. */
/* */
/* Constraints: load_aux(+) */
/* The constraints for the load_aux predicate are the same as the */
/* "load_wm" predicate. An additional constraint imposed in the */
/* "load_aux" predicate is that all OAV triples must be in the form */
/* of a Prolog list. */

load_aux(KB_Filename) :-
    !,
    convert_filename(KB_Filename,aux,Aux_Filename),
    see(Aux_Filename),
    loadfile,
    seen, !.

/*----- convert_filename -----*/
/*
/* The convert_filename predicate takes a filename and a new exten- */
/* sion and makes and returns a filename with the same prefix yet */
/* with the new extension. */
/* */
/* Constraints: convert_filename(+,+, -) */

convert_filename(KB_Filename, Ext, New_Filename) :-
    atom(KB_Filename),
    !,
    parse_name(KB_Filename, ASCII_PRE, ASCII_EXT),
    name(Ext, ASCII_EXT),
    append(ASCII_PRE, [46|ASCII_EXT], New_ASCII),
    name(New_Filename, New_ASCII), !.

/*----- parse_name -----*/
/*
/* The parse_name predicate takes an atom (which is usually a file- */
/* name) and divides it into two parts. The filename is parted into */
/* two ASCII lists, one representing the filename's prefix and the */
/* other representing the filename's extension. */
/* */
/* Constraints: parse_name(+,?, -) */

parse_name(Old_FN, ASCII_PRE, ASCII_EXT):-
    name(Old_FN, ASCII_List),
    member(46, ASCII_List),
    !,
    append(ASCII_PRE, [46], Temp_List),
    append(Temp_List, ASCII_EXT, ASCII_List).

```

```

/*----- makefile -----*/
/*
/* The makefile predicate will read in a term from a file, parse the
/* term and converts it so all quoted OAV triples are converted to
/* OAV triples within a Prolog list and then write them back out to
/* a second file. The term is read using the Prolog read predicate
/* and the outputting to the new file is done via ABC's change
/* predicate. The calling procedure is responsible for "seeing" and
/* "telling" the appropriate files.
/*
/* Constraints: The calling procedure must "see /1" a valid Prolog
/* formatted file to read from and must also "tell /1" a file to open
/* a output file. Note that if this file already exists,"tell" will
/* overwrite all old data in favor of the new data.
*/

```

```

makefile :-
    read(Term),
    change(Term).

```

```

/*----- change -----*/
/*
/* The change predicate controls the changing of the term read from
/* the ".kb" format to the ".abc" format & writes the new format out
/* to the new file with the new extension. For purposes of recur-
/* sive programming, it then calls makefile which will read another
/* term and the process is then repeated.
/*
/* Constraints: change(+)
/* Same constraints as the "makefile" predicate.
*/

```

```

change(end_of_file) :-
    !.

```

```

change(Term) :-
    convert_term(Term,NewTerm), /* Terms are parsed and converted via */
    write(NewTerm),             /* ABC's "convert_term /2" predicate. */
    write(' '),
    nl,
    makefile.

```



```

/*----- loadfile -----*/
/*
/* The loadfile predicate reads a term from a file and loads it in
/* working memory with the use of the load predicate. The load pred-
/* icate actually asserts the term into the working memory and
/* recursively calls loadfile until the end_of_file is seen and then
/* terminates the loading process. This predicate is the same as in
/* BC3.
/*
/* Constraints: The calling procedure must "see" a valid Prolog
/* file.

```

```

loadfile :-
    read(Term),
    load(Term).

```

```

/*----- load -----*/
/*
/* The load predicate asserts a term into the Prolog database and
/* calls the ABC predicate "loadfile" until there is no more terms
/* to load, at which time the "end_of_file" marker should be reached
/* and the load predicate will simply succeed not allowing back-
/* tracking.
/*
/* Constraints: load(+).
/* The ABC predicate "loadfile" must exist and its constraints met.

```

```

load(end_of_file) :-
    !.
load(Term) :-
    assertz(Term),
    loadfile.

```

```

/*----- load_abc -----*/
/*
/* The load_abc predicate checks to see if the filename provided is
/* an atom and then reconsults the file.
/*
/* Constraints: load_abc(+).
/* A valid filename must be provided as the input.

```

```

load_abc(Filename) :-
    atom(Filename),
    nl,nl,
    reconsult(Filename).

```

```

/*----- check -----*/
/*
/* The check predicate checks a file name for the proper extension. */
/* If the extension of the filename provided is either ".kb" or */
/* ".abc" the extension name is returned. Otherwise, the "check" */
/* predicate fails. */
/*
/* Constraints: check(+,-). */

check(File_Name,Type) :-
    atom(File_Name),
    name(File_Name, ASCII_List),
    check_ext(ASCII_List,Type).

/*----- check_ext -----*/
/*
/* The check_ext predicate is used in conjunction with the "check" */
/* predicate. The check_ext tries to match the extension of a file */
/* name with either "kb" or "abc" in the form of ASCII lists. */
/*
/* Constraints: check_ext(+,?). */
/* The first argument must be a list. */

check_ext([],[]) :-          /* If the filename is empty then the */
    !,                        /* predicate fails. */
    fail.

check_ext([46|T], kb) :-     /* The extension is ".kb" is true if */
    T = [107,98],            /* the filename's ASCII equivalent */
    !,                        /* list ends with [46,107,98] */

check_ext([46|T], abc) :-    /* The extension ".abc" is true if */
    T = [97,98,99],          /* the filename's ASCII equivalent */
    !,                        /* list ends with [46,97,98,99]. */

check_ext([_|T],List) :-     /* The extension hasn't been found */
    check_ext(T,List).        /* yet, recursively check the rest of */
                                /* the filename for the extension. */

/*----- convert_term -----*/
/*
/* convert_term(A,B) converts the user friendly term to one in which */
/* the ABC ES shell can understand. The convert_term predicate is */
/* used only on ABC data structures. It will eventually replace all */
/* the single quoted triples such as 'amy loves john' to a list */
/* structure such as [amy,loves,john] which is what the inference */
/* mechanism of ABC requires. */
/*
/* Constraints: convert_term(+,-). */
/* This predicate may not work unless altered on other than */
/* structures defined in ABC. */

```

```

convert_term(A : B, A : C) :-          /* A should be a key word */
    convert_term(B, C), !.             /* such as fact, askable, */
                                        /* etc. We break down */
convert_term(A then B, C then D) :-    /* the term until it */
    convert_term(A, C),                 /* cannot be broken */
    convert_term(B, D), !.             /* down (until its an atom) */
                                        /* any further. We token- */
convert_term(if A, if B) :-            /* ize the atom which is a */
    convert_term(A, B), !.             /* 0-A-V triple within sin- */
                                        /* gle quotes. The */
convert_term(A and B, C and D) :-      /* output from this */
    convert_term(A, C),                 /* step is the same */
    convert_term(B, D), !.             /* basic term except all */
                                        /* the 0-A-V triples */
convert_term(A or B, C or D) :-        /* in quotes are */
    convert_term(A, C),                 /* replaced by 0-A-V */
    convert_term(B, D), !.             /* triples in lists. */

convert_term(A slot B, C slot D) :-
    convert_term(A, C),
    convert_term(B, D), !.

convert_term(A derived_from B, C derived_from D) :-
    convert_term(A, C),
    convert_term(B, D), !.

convert_term(not A, not B) :-
    convert_term(A, B), !.

convert_term(A, List) :-                /* Here is where the atom */
    atom(A),                            /* gets converted. It will */
    name(A, OldList),                   /* only be converted if the */
    OldList = [First_Letter|Rest],
    ( First_Letter < 97, First_Letter > 64 ; member(32, Rest)),
    !,
    convert_triple(OldList, List), !.   /* atom is a Variable or if */
                                        /* the atom has a space in */
convert_term(A, A).                     /* it somewhere. */

/*----- convert_triple -----*/
/*
/* The convert_triple predicate will parse through a quoted atom and
/* tokenize the atom into a list of words. See the comments about
/* the predicates "tokenize /3" and "get_rest_word /4" for more
/* detailed explanation.
/*
/* Constraints: convert_triple(+,-)
/* The first argument must be a list of ASCII numbers.
*/

convert_triple(OldList, List) :-
    tokenize(OldList, [], List).

```

```

/*----- tokenize -----*/
/*
/* The tokenize predicate receives as an argument, a list of ASCII
/* numbers representing the O-A-V triple which was in quotes. This
/* list is then tokenized into a list of words. Tokenize does this
/* through the aid of the get_rest_word predicate. Tokenize gets
/* all the words from within the quoted atom A which are separated
/* by a space, appends them to the list called List, and forms the
/* new list called NewList. The tokenize predicate is an alteration
/* of a predicate by the same name taken from Claudia Marcus's book
/* "Prolog Programming", pages 203-210.
/*
/* Constraints: tokenize(+,+,+,-).
/* The first argument must be a list of ASCII numbers. The second
/* argument must be a list.

tokenize([H|T], List, L) :-          /* Make certain that the first
    identifier(H),                  /* thing in the quoted list is
    !,                              /* an identifier, then get the
    get_rest_word(T,[H],Word,Rem),  /* rest of the word. Once all
    append(List,[Word],NewList),    /* the word is obtained, append
    tokenize(Rem,NewList,L).        /* it to "List" to get "NewList".*/
                                   /* Then tokenize the rest of the
tokenize([32|T], List, L) :-        /* List. Remove all nonessential
    !,                              /* blanks.
    tokenize(T, List, L).

tokenize([], List, List).

tokenize([39|T],List,L) :-          /* Quoted atoms within the quotes*/
    !,                              /* for the OAV triple must be
    get_rest_quote(T,[39],Word,Rem), /* treated differently. A pred-
    append(List,[Word],NewList),    /* cate called "get_rest_quote"
    tokenize(Rem,NewList,L).        /* is used to get the rest of the
                                   /* quote.

/*----- get_rest_quote -----*/
/*
/* The get_rest_quote predicate gets as its first argument a list of
/* ASCII numbers representing the remaining portion of an OAV triple
/* which it is trying to tokenize. The leading quote has already
/* been seen and now get_rest_quote will get the remaining portion of
/* the quote, convert it from an ASCII list to an atom, and return
/* it as its third argument.
/*
/* Constraints: get_rest_quote(+,+,+,-,-).
/* The first argument must be a list of ASCII numbers. The second
/* argument must be a list.

```

```

get_rest_quote([39|T], List, Word, T) :- /* If a quote mark is ob- */
!,                                         /* served, append it to */
append(List,[39],NewList),               /* the end of the list rep-*/
name(Word,NewList).                      /* resenting the rest of */
                                         /* quote and convert it to */
                                         /* an atom and return it. */

get_rest_quote([H|T], List, Word, Rem) :- /* Otherwise, append the */
!,                                         /* next character of the */
append(List, [H], NewList),              /* quote to the end of the */
get_rest_quote(T, NewList, Word, Rem).    /* list and continue. */

/*----- get_rest_word -----*/
/*
/* The get_rest_word is fundamentally the same as the "get_rest_
/* quote" predicate. The get_rest_word predicate will be given an
/* ASCII list L1, and a second list L2, which may have ASCII num-
/* bers also, and will append all the ASCII numbers up to the first
/* '32', representing a space, of L1 to L2, convert it over to text
/* and output it as Word and output the remainder of the ASCII
/* numbers as REM.
/*
/* Constraints: get_rest_word(+,+,--,-).
/* Same as get_rest_quote.
*/

get_rest_word([H|T], List, Word, X) :- /* Get the next ASCII num- */
identifier(H),                          /* ber and make sure it rep-*/
!,                                       /* resents an identifier. */
append(List, [H], NewList),            /* Append the ASCII number */
get_rest_word(T, NewList, Word, X).     /* to a temporary list and */
                                         /* continue this operation */
                                         /* until a space mark is */
                                         /* encountered. When a "32"*/
                                         /* (a space) is seen, re- */
                                         /* turn the converted list */
                                         /* along with what remains */
                                         /* to be parsed.
get_rest_word([32|T], List, Word, T) :- /* When there is no more */
name(Word, List),                       /* ASCII list to be token- */
!.                                       /* ized, the list is con- */
                                         /* verted to a word and */
                                         /* returned.
get_rest_word([], List, Word, []) :-
!,
name(Word, List).

/*----- identifier -----*/
/*
/* The identifier predicate identifies valid characters which may be
/* inside ABC's OAV triples.
/*
/* Constraints: identifier(+).
/* The argument must be an atom.
*/

```

```

identifier(D) :-      /* An identifier can be any letter or digit.      */
    letter(D);
    digit(D),
    !.

identifier(95).      /* An identifier can also be an underscore.      */
identifier(36).      /* An identifier can also be a $ sign.            */
identifier(46).      /* An identifier can also be a period.            */
identifier(63).      /* An identifier can also be a question mark.     */

/*----- letter -----*/
/*
/* The letter predicate tests to see if an ASCII number symbolizes
/* some letter of the alphabet. If it does, the "letter" predicate
/* succeeds. If not, the predicate fails.
/*
/* Constraints: letter(+).
/* The argument must be an ASCII number between 1 and 255.

letter(A) :-      /* A letter is any capital letter between
    A =< 90,      /* "A" (ASCII #65) and "Z" (ASCII #90).
    A >= 65,
    !.

letter(B) :-      /* Or a letter is any small letter between
    B =< 122,     /* "a" (ASCII #97) and "z" (ASCII #122).
    B >= 97,
    !.

/*----- digit -----*/
/*
/* The digit predicate tests to see if an ASCII number symbolizes
/* one of the digits between and including zero through nine. The
/* predicate succeeds if it does symbolize a digit and fails if it
/* does not symbolize a digit.
/*
/* Constraints: digit(+). Same as the predicate "letter".

digit(C) :-      /* A digit is any ASCII character between
    C =< 57,      /* 48 and 57 (Zero through Nine).
    C >= 48.

```

```

/*----- consultation -----*/
/*
/* The ABC command CONSULTATION is similar to BC3's START predicate. */
/* It initializes the KB why retracting all the why_traces, reversing */
/* the goal list and inserting the reversed goal list in the trace */
/* mechanism. This is also the top level predicate which tries to */
/* solve the goals. */

execute(consultation) :-
    cls,                                /* This predicate starts by clear- */
    retract_all(why_trace(_)),          /* the screen and the why-trace */
    ask_initial_askables,              /* mechanism. It then asks the user*/
    goals : Goals,                     /* all askables declared initial. */
    solve(Goals),                      /* It then solves the goals and */
    reverse_goals(Goals,Rev_Goals),    /* asserts the goals into the trace*/
    assert_goals(Rev_Goals),          /* mechanism in reversed order. */
    nl,nl,nl,
    write('Do you wish to save working memory to disk? (<y>,n) '),
    get_reply(Reply),
    ( Reply = yes,                      /* This predicate also prompts the */
      save_vm                          /* user to inquire about saving */
    ;                                  /* working memory. If answered */
      true                             /* with a positive response, the */
    ), nl,nl,                          /* predicate "save_vm" is executed */
    !,                                /* otherwise, the predicate con- */
    fail.                             /* tinues through the cut and */
                                     /* ultimately failing causing a */
                                     /* return to the ABC prompt. */

execute(consultation) :-              /* If the goals cannot be solved, */
    nl,nl,tab(16),
    write('ABC Consultation Complete - No Solutions Found'),
    !,
    fail.                             /* then the "no solutions" message */
                                     /* is posted prior to returning to */
                                     /* the ABC prompt. */

/*----- go -----*/
/*
/* Allows the user to type "go" as an ABC command as opposed to the */
/* ABC command "consultation." */

execute(go) :-                       /* "go" simply executes the ABC com- */
    !,                                /* mand called "consultation". */
    execute(consultation).

```

```

/*----- ask_initial_askables -----*/
/*
/* The ask_initial_askables predicate searches the Prolog database */
/* for "initial_askables" and if found, asks the user a question and */
/* prompts him for a reply. This will continue until all "initial_ */
/* askables" are asked. */
/*
/* Constraints: All initial_askables in the Prolog database must be */
/* structured in accordance to the ABC user's manual. */

```

```

ask_initial_askables :-
    initial_askable : [Obj,Attr,Val] derived_from Question and
Valid Answers,
    ask_question([Obj,Attr,Val],Question,Valid Answers,Value,CF),
    assert_in_trace(askable : [Obj,Attr,Value,CF]),
    fail.
ask_initial_askables.

```



```

/*----- ask_question -----*/
/*
/* The ask_question predicate checks to see if the question can be
/* asked. If the OAV triple associated with the question is already
/* in the trace, then the question will not be allowed. The ask
/* question predicate sets up how the question is asked, gets the
/* user's reply to the question, and asserts the appropriate OAV
/* triple in the trace mechanism. It also allows the user to in-
/* quire as to "why" a question is being asked.
ask_question([Obj,Attr,Val],Question,Valid_Answers,Value,CF) :-
    not_in_trace(Obj,Attr),      /* Checks to see if the object-attr
    nl,nl,nl,                    /* pair is in the trace.
    name(Query,Question),
    write(Query),nl,
    write_valid_answers(Val,Valid_Answers,1,Nbr_of_Answers),
    get_answer(Nbr_of_Answers,Answer),!,
    ( Answer = why,
      clean_up_why_trace(Obj,Attr,Val), /* Delete remnants of failed
      explain_why([Obj,Attr,Val]),      /* rules from the why trace.
      ask_question([Obj,Attr,Val],Question,Valid_Answers,Value,CF),
      !
    ;
      /* Ask the question.
      Valid_Answers = [yes | _],
      nonvar(Val),
      Value = Val,
      ( Answer = 1, !,
        CF = 100, !
      ;
        CF = 0, !
      )
    ;
      /* If the valid answers are a numer-
      Valid_Answers = [Value|_], /* list and the user replied with a
      CF = 100, !                /* one, then assert the head of the
      ;                          /* valid-answers list into the trace.
      Structure =.. Valid_Answers, /* Otherwise, turn the valid answer
      Arg is Answer - 1,          /* list into a structure and use the
      arg(Arg,Structure,Value),   /* "arg" predicate to acquire the
      CF = 100                    /* correct corresponding answer.
    ), !.

```

```

/*----- write_valid_answers -----*/
/*
/* The write_valid_answers predicate is responsible for numerating
/* and displaying all the valid answers in a "pretty" format. It
/* also acquires the number of valid answers and passes this infor-
/* mation on so a check can be made to make sure the user doesn't
/* try to select a number that does not correspond to a valid choice.
/*
/* Constraints: write_valid_answers(+,+,+,-).
/* The first argument must be an atom. The second argument must be
/* a list. The third argument must be an integer.
*/

```

```

write_valid_answers(, [], 0) :- !.
write_valid_answers(Value, [Valid Answer|Rest], Answer_Nbr, Max_Nbr) :-
    tab(4), write(Answer_Nbr), write(' : '), write(Valid Answer),
    ( Rest = [], !,
      Max_Nbr = Answer_Nbr, nl      /* If no more values exist in valid
      ;                               /* answer list, the number of answers
      Next_Number is Answer_Nbr + 1, /* is equal to the last answer's
      nl,                               /* number. Otherwise, recur-
      write_valid_answers(Value,      /* sively loop to get next number
      Rest, Next_Number, Max_Nbr)    /* while incrementing Answer number.
    ), !.

```

```

/*----- get_answer -----*/
/*
/* The get_answer predicate is given as its first argument, the max-
/* imum number which can be returned as a valid response to an
/* askable. If the user tries to respond with a invalid answer, the
/* get_answer predicate will flag the "invalid input" and prompt the
/* user for a valid answer. This predicate returns either the num-
/* which corresponds to answer selected by the user or the atom "why"
/* which corresponds to the user wanting an explanation to the
/* question.
/*
/* Constraints: get_answer(+,-).
/* The first argument must be an integer between 1 and 9.
*/

```

```

get_answer(Max_Nbr, Answer) :-
    nl, write('Enter Number or w (for why) > '),
    get(Reply), !,
    ( Reply = 119,                               /* After the user is prompted to pro-
      Answer = why, !                             /* an answer, the user responds by
      ;                                           /* typing in the letter "w" (ASCII
      ( Reply < 49                               /* number 119) or a number between one
      ; Reply > 48 + Max_Nbr                     /* and the max number.
      ), nl, write('Invalid Input.'), /* The ASCII number for one is
      write(' Please try again.'), nl, /* 49.
      get_answer(Max_Nbr, Answer), !
      ;
      Answer is Reply - 48
    ), !.

```

```

/*----- solve -----*/
/*
/* The predicate solve is used to solve "Goals" in the form of
/* "goals : goal_1 and goal_2 and ... goal_n." where each goal is a
/* triple. The solve predicate also initiates the "why" trace and
/* displays all solved triples which are not explicitly told by the
/* user.
/*
/*
/* Constraints: solve(+).
/* Goals must take the form outlined in the ABC user's manual.
*/

```

solve([]).

solve(Goals) :-

```

( Goals = [Obj,Attr,Val] and Others /* Multiple goals are split. */
; /* Single goals require that */
Goals = [Obj,Attr,Val], /* "Others" be instantiated to*/
Others = [] /* an empty list. The first */
), /* goal is solved first then */
retract_all(why_trace(_)), /* the "Others" are solved. */
asserta(why_trace(goal:[Obj,Attr,Val])),!, /* Initiate why-trace. */
known([Obj,Attr,Val],CF), /* Solution requires that the */
!, /* OAV triple is "known" and */
( CF = 100, /* instantiated. If a value */
List = [Val cf 100] /* of a O-A tuple is "known" */
; /* with a CF of 100; quit and */
get_list(Obj,Attr,Val,List) /* return that one value, */
), /* otherwise, get a list of */
write_goal(Obj,Attr,Val,List),!, /* all the values and write */
nl,nl,tab(27), /* out the list. */
write('Hit any key to continue.'),
get0(_), /* Solve the rest of the */
solve(Others). /* goals. */

```

```

/*----- known -----*/
/*
/* The known predicate is activated when a top-level rule or frame
/* matches a goal. It attempts to satisfy this top-level rule or
/* frame by matching it against other facts, assertions, rules,
/* frames or askables in the knowledge base. If the top level rule
/* or frame is "known", then the goal is said to have a solution.
/* The "known" predicate uses the "is_known" predicate to solve the
/* lower levels of the search for a solution. If a triple is "known"*/
/* the "known" predicate will succeed and will return a calculated
/* certainty factor giving the relative strength of how well the OAV
/* triple is "known".
/*
/*
/* Constraints: known(+,-).
/* The first argument must be a OAV triple.
*/

```

```

known([Obj,Attr,Val], CF) :-      /* The goal matches the conclusion of */
    Rule : if Conds then [Obj,Attr,Val|Rule_CF],          /* a rule.      */
    retract_all(rule_trace(Rule,_,_)),
    clean_up_why_trace(Obj,Attr,Val),
    asserta(why_trace(Rule : [Obj,Attr,Val|Rule_CF] derived_from Conds)),
    is known(Conds, Rule, 1, Cond_CF),
    ( ( Rule_CF = []                /* The overall CF is the product of */
      ; Rule_CF = [cf,100]          /* rule CF and the lowest CF of the */
    ),                             /* conditions divided by 100, so if */
      CF = Cond_CF                  /* rule CF is 100, then this is re- */
    ;                               /* duced to the overall CF equalling */
      Rule_CF = [cf,CF1],           /* the CF of the conditions.        */
      CF is Cond_CF * CF1 / 100     /* Otherwise, perform the calculations*/
    ),                             /* and assert the rule and its cond- */
    assert_in_trace(Rule),          /* itions in the trace mechanism.    */
    assert_in_trace(Rule : [Obj,Attr,Val,CF] derived_from Conds),
    CF = 100,!.                    /* If the overall CF is less than 100,*/
                                   /* backtrack and get other solutions */
                                   /* if they exist.                    */

known([Obj,Attr,Val], CF) :-      /* The goal matches a value being */
    frame : Obj slot Slots,        /* stored in a frame.              */
    frame_get(Obj,Attr,Value_List),
    member(Val,Value_List),
    assert_in_trace(frame_fact : [Obj,Attr,Val,100]),
    CF = 100,!.

known([Obj,Attr,Val], CF) :-      /* There were no solutions to */
    ( trace(_ : [Obj,Attr,Val | CF1]) /* goal which had a CF of 100 so */
    ;                               /* the overall solution to the */
      trace(Rule : [Obj,Attr,Val | CF1] derived_from _ ) /* goal is the */
    ),                             /* individual group of solutions */
    ( ( CF1 = [] ; CF1 = [100]),      /* which were asserted into the */
      CF = 100                       /* trace.                        */
    ;
      CF1 = [CF]
    ).

/*----- assert_in_trace -----*/
/*
/* The assert_in_trace predicate asserts one of the ABC structures
/* into the trace. If it already exists in the trace, then it will
/* not be asserted a second time. If a fact, confirmed assertable,
/* or a rule already exists with the same OAV triple as a solution,
/* then the original OAV triple's CF will be adjusted to reflect the
/* additional support but the new structure will not be placed in
/* the trace. If neither of the above is true, only then will the
/* structure be placed into the trace mechanism.
/*
/* Constraints: assert_in_trace(+).
/* The argument must be a valid ABC structure (i.e. fact: [O,A,V]),
/* or the rule number of a top level rule which solved the goal.
*/

```

```

assert_in_trace(Head : Triple) :-      /* Do not assert a clause in the */
    trace(Head : Triple),              /* trace mechanism if it already */
    !.                                  /* exists.                        */

assert_in_trace(Rule : [Obj,Attr,Val|CF1] derived_from Conds) :-
    ( trace(Head : [Obj,Attr,Val|CF2]),
      calculate_CF(CF1,CF2,CF),
      replace_trace(Head : [Obj,Attr,Val,CF])
    ;
      trace(Head : [Obj,Attr,Val|CF2] derived_from Cond_2),
      calculate_CF(CF1,CF2,CF),
      replace_trace(Head : [Obj,Attr,Val,CF] derived_from Cond_2)
    ;
      asserta(trace(Rule : [Obj,Attr,Val|CF1] derived_from Conds))
    ), !.

assert_in_trace(Head : [Obj,Attr,Val|CF1]) :-
    ( trace(Old_Head : [Obj,Attr,Val|CF2]),
      calculate_CF(CF1,CF2,CF),
      replace_trace(Old_Head : [Obj,Attr,Val,CF])
    ;
      trace(Rule : [Obj,Attr,Val|CF2] derived_from Conds),
      calculate_CF(CF1,CF2,CF),
      replace_trace(Rule : [Obj,Attr,Val,CF] derived_from Conds)
    ;
      asserta(trace(Head : [Obj,Attr,Val|CF1]))
    ),
    !.

assert_in_trace(Rule_No) :-              /* If a rule is being placed in */
    retract(rule_trace(Rule_No, _, Clause)), /* the trace mechanism, */
    assert_in_trace(Clause),              /* then also place the proven */
    fail.                                  /* conditions of the rule in the */
assert_in_trace(_) :- !.                  /* trace also.                */

/*----- calculate_CF -----*/
/*
/* The calculate_CF predicate takes as its two inputs, two certainty */
/* factors, which it uses to calculate the combining certainty */
/* factor from. It uses the formula  $CF = CF1 + (100 - CF1) / 100 * CF2$ . */
/*
/* Constraints: calculate_CF(+,+, -). */
/* The two inputs must be either lists, empty or with one number */
/* equal to or less than 100, or they may be just one number equal */
/* to or less than 100. The two types cannot be mixed. */

calculate_CF([],_,100).                  /* When one of the certainty factors */
calculate_CF([100],_,100).               /* is known with a CF of 100, then it */
calculate_CF(100,_,100).                 /* doesn't matter what the other one */
calculate_CF(_,[],100).                  /* is, the overall CF will always be */
calculate_CF(_,[100],100).               /* 100. */
calculate_CF(_,100,100).

```

```

calculate_CF([CF1],[CF2],CF) :-          /* Calculate the overall */
    CF is CF1 + (100 - CF1) / 100 * CF2, !. /* CF by using the for- */
calculate_CF(CF1,CF2,CF) :-              /* mula described above. */
    CF is CF1 + (100 - CF1) / 100 * CF2, !.

```

```

/*----- replace_trace -----*/
/*
/* The replace_trace predicate replaces a clause in the trace mech-
/* anism with an updated clause. Actually all that gets updated is
/* the certainty factor. Whenever a second rule or fact derives a
/* a solution which has already been found and placed in the trace,
/* the second rule or fact will not be asserted, yet its certainty
/* factor will be used to increase the CF of the original assertion.
/*
/* Constraints: replace_trace(+).
/* The argument must be a valid ABC knowledge structure.
*/

```

```

replace_trace(Rule : [Obj,Attr,Val,CF] derived_from Conds) :-
    assertz(trace(marker)),
    circulate_trace(Rule : [Obj,Attr,Val,CF] derived_from Conds),
    !.

```

```

replace_trace(Head : [Obj,Attr,Val,CF]) :-
    assertz(trace(marker)),
    circulate_trace(Head : [Obj,Attr,Val,CF]),
    !.

```

```

/*----- circulate_trace -----*/
/*
/* The circulate_trace predicate actually does all the work which
/* the "replace_trace" predicate is responsible for. It circulates
/* the trace clauses in the Prolog database, "popping off" clauses
/* from the top of the stack and placing them back on the bottom,
/* until the entire stack of trace structures have been moved back
/* to their original position. Additionally, it replaces the CF of
/* the old trace structure with the new CF in the shuffle.
/*
/* Constraints: circulate_trace(+).
/* The argument must be a valid ABC knowledge structure.
*/

```

```

circulate_trace(Rule : [Obj,Attr,Val,CF] derived_from Conds) :-
    retract(trace(Knowledge)),          /* If knowledge structure is a */
    ( Knowledge \= marker,              /* rule matching the argument, */
      ( Knowledge = Rule : [Obj,Attr,Val] derived_from Conds,
        assertz(trace(Rule : [Obj,Attr,Val,CF] derived_from Conds))
      ;
        assertz(trace(Knowledge))      /* then change its CF and assert */
      ),!,                             /* it back, otherwise just */
      circulate_trace(Rule : [Obj,Attr,Val,CF] derived_from Conds), /* assert the rule back. */
      !,
      ;
        true                           /* If the marker has not been */
      ),!.                             /* seen, continue circulating the */
                                     /* trace, otherwise quit. */

circulate_trace(Head : [Obj,Attr,Val,CF]) :-
    retract(trace(Knowledge)),          /* If the knowledge structure is */
    ( Knowledge \= marker,              /* something other than a rule, */
      ( Knowledge = Head : [Obj,Attr,Val] , /* then this rule applies.*/
        assertz(trace(Head : [Obj,Attr,Val,CF]))
      ;
        assertz(trace(Knowledge))
      ),
      circulate_trace(Head : [Obj,Attr,Val,CF]),
      !,
      ;
        true
      ),!.

/*----- get_list -----*/
/*
/* The get_list predicate assumes that Obj-Attr-Val triple is in the */
/* trace somewhere with a certainty factor less than 100. The get_ */
/* list predicate will return a list of all values along their */
/* certainties for every different value found in the trace which */
/* matches the Obj and Attr. */
/*
/* Constraints: get_list(+,+,+,-). */
/* The three arguments provided must all be atoms. */

```

```

get_list(Obj,Attr,Val,List) :-
    nonvar(Val),                /* Find all knowledge structures in */
    retract_all(temp_list(_)), /* trace which matches the Obj-Attr */
    asserta(temp_list([])),     /* pair and place them in a list. */
    !,
    ( ( trace(Type : [Obj,Attr,NewVal,CF1]),
        Type \= confirmed_not,
        CF1 \= 0
      ;
        trace(_ : [Obj,Attr,NewVal,CF1] derived_from _),
        CF1 \= 0
      ),
      temp_list(Partial_List), /* If a triple is already*/
      not_member(NewVal_cf_, Partial_List), /* part of the list, do */
      Temp_List = [NewVal_cf CF1 | Partial_List], /* not add, other- */
      retract_all(temp_list(_)), /* wise, assert into the */
      asserta(temp_list(Temp_List)), /* list, fail, and go */
      fail, /* backtracking to get */
      ; /* the rest. */
      retract(temp_list(List)),
      retract_all(temp_list(_))
    ), !. /* When you get all the */
        /* triples, return the */
        /* List. */

```

```

/*----- write_goal -----*/
/*
/* When a goal is solved, this fact is displayed on the screen with */
/* all the solutions (if no solution has a certainty factor of 100%) */
/* except those goals which were directly obtained and confirmed by */
/* asking the user. */
/*
/* Constraints: write_goal(+,+,+,+). */

```

```

write_goal(Obj,Attr,Val,List) :-
    cls,
    nl,nl,nl,
    tab(17),
    write('The Solution(s) To The Goal Are Listed Below'),nl,tab(17),
    write('====='),nl,nl,
    write_all_goals(Obj,Attr,Val,List), !.

```

```

write_all_goals(_,_,_,[]).
write_all_goals(Obj,Attr,_,[Val cf CF|Rest]) :-
    ( confirmed([Obj,Attr,Val]),!
    ;
        nl,tab(15),
        writelist([Obj,Attr,Val,with,cf,of,CF,'.']),
        nl
    ), !,
    write_all_goals(Obj,Attr,_,Rest).

```



```

/*----- reverse_goals -----*/
/*
/* The reverse_goals predicate simply reverses the goals list so
/* that the goals may be asserted into the trace mechanism in their
/* proper order. Because of operator precedence problems, the
/* reversed goals (Reversed_Goals) must be made up of the last goal
/* found in the original goal list with the ABC operator "and" and
/* the reverse of the remainder of the original goals.
/*
/* Constraints: reverse_goals(+,-).

reverse_goals(Goal and Rest, Last_Goal and Rev_Rest) :-
    get_last_goal(Goal and Rest, Last_Goal, Remainder),
    reverse_goals(Remainder, Rev_Rest), !.
reverse_goals(Goal, Goal).

/*----- get_last_goal -----*/
/*
/* The get_last_goal predicate simply breaks the goals list into the
/* the last goal and all the remaining goals.
/*
/* Constraints: get_last_goal(+,-,-).

get_last_goal(Goal1 and Goal2 and Rest, Last_Goal, Goal1 and Rem) :-
    get_last_goal(Goal2 and Rest, Last_Goal, Rem), !.
get_last_goal(Goal1 and Goal2, Goal2, Goal1).
get_last_goal(Goal, Goal, _).

/*----- assert_goals -----*/
/*
/* The "assert_goals" predicate assert goals into trace mechanism
/* only if all the goals can be solved.
/*
/* Constraints: assert_goals(+).

assert_goals(Goal and Other_Goals) :-
    assert_trace(goal : Goal),          /* Assert the head goal and then */
    assert_goals(Other_Goals).          /* recursively get the rest of   */
assert_goals([Obj,Attr,Val]) :-        /* goals and assert them also.  */
    assert_trace(goal : [Obj,Attr,Val]).
assert_goals(_).

/*----- save_wm -----*/
/*
/* The save working memory predicate saves the working memory por-
/* tion of the knowledge base to a separate file. The filename is
/* provided by the user. The file can easily be read and altered by
/* an ASCII editor. The save_wm predicate will save all the frames
/* along with all the confirmed and denied triples in the working
/* memory of the Prolog database.
/*
/* Constraints: None

```

```

save_wm :-
    cl̄s,
    nl,nl,nl,
    write('You have elected to save working memory to disk. '), nl,
    write('Please supply a filename or press <RETURN> to abort. '), nl,nl,
    write('ABC save_wm > '),
    readline([],Filename),          /* The user either provides a file- */
    ( Filename = '',                /* name or aborts the procedure. */
      nl, write('Procedure to save working memory is aborted. '),nl,nl
    );
    tell(Filename),                /* If a filename was given, the file */
    write('/* '), nl,              /* is opened an a header is printed. */
    write('This is the working memory file for the file      called
'),
    write(Filename), nl,          /* Another header for the frames is */
    write(' /* '),nl,nl,nl,      /* written to the file. */
    write('/* Below is a listing of all the frames.          /* '),
    nl,nl,
    write_frames,                /* The frames are written to the file.*/
    nl,nl,
    write('/* Below is a listing of all confirmed triples.    /* '),
    nl,nl,                      /* A header for the confirmed triples */
    write_confirmed_triples,    /* is written followed by the triples */
    nl,nl,                      /* themselves. */
    write('/* Below is a listing of all denied triples.      /* '),
    nl,nl,
    write_denied_triples,       /* Last, the denied triples are sent */
    told                        /* to the file and the file is closed.*/
    ).

/*----- write_frames -----*/
/*
/* The write_frames predicate retrieves all of the frames in working */
/* memory portion of the Prolog database and writes them to the */
/* current output device. */
/*
/* Constraints : None */

write_frames :-
    frame : Frame slot Slots,
    write('frame : '),
    write(Frame),nl,
    write_slots(Slots),
    nl,
    fail.
write_frames.

```

```

/*----- write_slots -----*/
/*
/* The write_slots predicate takes all the slots of a given frame */
/* "pretty prints" them to the current output device. */
/*
/* Constraints: write_slots(+). */
/* Argument must be in the form "[Slot,SAttr,SVal, ...] slot [...]" */

write_slots(Slot slot Slots) :-      /* If there is more than one slot */
!,                                   /* then print out the word "slot" */
write(' slot '),                     /* followed by a single quote. */
writelist(Slot),put(39),nl,!,        /* Print out the slot list followed*/
write_slots(Slots).                  /* by another single quote. */

write_slots(Slot) :-                 /* The last slot goes through the */
!,                                   /* same procedure as above however,*/
write(' slot '),                     /* a period is placed after it */
writelist(Slot),put(39),             /* to indicate that its the end of */
write('.'), nl, !.                  /* term when read by the Prolog */
/* reader. */

/*----- write_confirmed_triples -----*/
/*
/* The write_confirmed_triples predicate is used by the "save_wm" */
/* predicate to save all the confirmed triples to working memory. */
/*
/* Constraints: None */

write_confirmed_triples :-           /* Retrieve confirmed triples */
confirmed(Triple),                  /* from the Prolog database and */
writeq(confirmed(Triple)), write('.'), nl, /* them to the current */
fail.                               /* output device. Backtrack and */
write_confirmed_triples.            /* the other confirmed triples. */

/*----- write_denied_triples -----*/
/*
/* The write_denied_triples predicate is used by the "save_wm" pred- */
/* icate to save all the denied triples to working memory. */
/*
/* Constraints: None */

write_denied_triples :-              /* Retrieve denied triples from */
denied(Triple),                     /* the Prolog database and write */
writeq(denied(Triple)), write('.'), nl, /* them to the current */
fail.                               /* output device. Backtrack and */
write_denied_triples.               /* get the other denied triples. */

```

```

/*----- restore -----*/
/*
/* The ABC command RESTORE, removes all the asserted knowledge which */
/* was "learned" during a consultation. It also destroys the trace */
/* of past consultations. It does this through the use of two similar*/
/* predicates, RESTORE_KB (which does the retractions) and RESTORE_KB */
/* WC (restore the KB with comments) which prompts the user to see if */
/* he would like to abort the operation. */
/* */
/* Constraints: None */

execute(restore) :-
    restore_kb_wc,
    !,
    fail.

restore_kb_wc :-
    /* Write out a warning to the user. */
    nl,nl,
    write('CAUTION: You are attempting to restore your knowledge base'),
    nl,
    write('This will delete all knowledge learned during past
consultations'),
    nl,
    write('from working memory.'),
    nl,nl,
    write('Do you wish to continue with the restore operation? '),
    get_reply(Reply),
    ( Reply = yes,
        /* If the user chooses to continue, */
        restore_kb /* then restore the knowledge base. */
    ;
        true
    ).

/*----- restore_kb -----*/
/*
/* The restore_kb predicate retracts all the temporary knowledge of */
/* previous consultations which were inserted into the Prolog data- */
/* base. */
/* */
/* Constraints: None */

restore_kb :-
    retract_all(why_trace(_)),
    retract_all(trace(_)),
    retract_all(rule_trace(_,_,_)),
    retract_all(confirmed(_)),
    retract_all(denied(_)).

```

```

/*----- trace -----*/
/*
/* The trace mechanism is very different in ABC than in the BC3 shell.*/
/* Rather than keeping the trace in the form of a list which was */
/* passed as a parameter, ABC uses Prologs built-in Top-Down structure*/
/* and asserta/assertz asserting mechanisms to place the pieces of the*/
/* trace into memory. It then uses Prolog's built-in search mechanism*/
/* to extract the information. This is an improvement, especially in */
/* larger KBs where the lists may get large and unmanageable very */
/* quickly. If no trace exists, "trace(X)" will fail and the message */
/* that no trace exists will appear. */

execute(trace) :-
    trace(X),          /* Obtain the next item in the trace */
    cls,              /* by matching the variable "X" to it.*/
    nl,nl,nl,         /* If one does not exist, then fail */
    tab(26),          /* go to the trace rule below. */
    write('ABC Trace Since Last Restore'), /* Write a header on the */
    nl,tab(26),        /* screen. */
    write('====='),
    nl,nl,
    !,
    print_trace,       /* Print the trace out to the current */
    !,                /* output device. */
    fail.              /* Fail and go back to ABC prompt. */

execute(trace) :-
    nl,nl,
    tab(22),
    write('There is NO trace in working memory.'),
    nl,
    !,
    fail.

/*----- print_trace -----*/
/*
/* The print_trace predicate actually lists the trace on the screen. */
/* It extracts the top trace of Prolog database stack and matches */
/* it to either a goal, an askable, a fact, etc and writes the trace */
/* on the screen preceded by one or more words to show the user how */
/* the triple was placed into the trace. The last line, "get0(13)", */
/* allows one step of the trace to be shown at a time so the trace */
/* will not scroll off the screen. Whenever the user presses the */
/* space bar, the next triple in the trace will be shown. This loop */
/* continues until no more triples are left or until the user */
/* presses the ENTER key. */
/*
/* Constraints: None */

```

```

print_trace :-
    trace(X),
    ( nl,
      X = (goal : Triple),
      write('Goal: '),          /* Print out a goal.          */
      writelist(Triple),
      nl
    ;
      X = (askable : Triple),
      write('Askable: '),       /* Print out an askable.    */
      writelist(Triple),
      nl
    ;
      X = (fact : Triple),
      write('Fact: '),          /* Print out a Fact.        */
      writelist(Triple),
      nl
    ;
      X = (solved : Triple),
      write('Solved: '),        /* Print out triples which were */
      writelist(Triple),        /* mathematically solved.      */
      nl
    ;
      X = (was_told : Triple),
      write('Told: '),          /* Print out triples which were told */
      writelist(Triple),        /* to ABC explicitly by the user.  */
      nl
    ;
      X = (confirmed_not : Triple),
      write('Contradicted: '),  /* Print out triples which were */
      writelist(Triple),        /* contradicted.                */
      nl
    ;
      X = (was_proved : [Triple,Rule]),
      write('Proved: '),
      writelist(Triple),        /* Print out triples which were */
      write(' using '),         /* proved solving a rule.       */
      write(Rule),
      nl
    ;
      X = (Rule : Triple derived_from Conditions),
      write(Rule),
      write(': '),
      writelist(Triple),
      write(' was derived from'),
      nl,
      write_conditions(Conditions),
      nl
    ;

```

```

X = (frame_fact : [Obj,Attr,Val|Rest]),
write('From the frame '),
write(Obj),          /* Print out triples found from */
write(', '),          /* frames. */
writelst([Obj,Attr,Val]),
write(' was obtained. '),
nl
;
X = (via_ako : [Frame,Parent Frame]),
write('Via an ako link from '),
write(Frame),        /* Print out "ako" links between */
write(' to '),        /* frames when these links are used. */
write(Parent_Frame),
write(', '),
nl
),
get0(13).

/*----- remove_kb -----*/
/* */
/* This ABC command, REMOVE KB, removes a KB from working memory. */
/* Because this is such a drastic step, the user is prompted to see if */
/* he wishes to continue. If continued, all rules, facts, frames, and */
/* askables will be removed from working memory. */
/* */
/* Constraints: None. */

execute(remove_kb) :-
  nl,nl,
  write('CAUTION: You are attempting to remove your knowledge base'),
  nl,
  write('from working memory. '),
  nl,nl,
  write('Do you wish to continue with the removal operation? '),
  get_reply(Reply),
  ( Reply = yes,
    remove_kb
  ;
    nl,nl,
    write('Aborting remove knowledge base operation. '),
    nl
  ),
  !,
  fail.

```

```

remove_kb :-
    restore_kb,
    retract_all(goals : _),
    retract_all(fact : _),
    retract_all(askable : _ derived_from _ and _),
    retract_all(initial_askable : _ derived_from _ and _),
    retract_all(Rule : if _ then _),
    retract_all(frame: _ slot _).

/*----- review_goals -----*/
/*
/* This ABC command, REVIEW_GOAL, allows the user to display the cur- */
/* rent goal list. */

execute(review_goals) :-
    cls,
    nl,nl,nl,
    tab(23),
    write('Goals Currently in Working Memory'),
    nl, tab(23),
    write('====='),
    nl,nl,
    ( goals : Goals,
      print_goals(Goals)
    ;
      tab(29),
      write('No Goals Can Be Found') ),
    !,
    fail.

/*----- import -----*/
/*
/* The import command is similar to the load commands in that it */
/* loads in (via Prolog's consult predicate) a file from disk. It */
/* differs in that it loads in auxiliary files and/or demon files. */

execute(import) :-
    nl,nl,
    write('This ABC command imports Prolog code into working memory. '),nl,
    write('Do you wish to continue? (<y>,n): '),
    get_reply(Rply),nl,nl,! ,
    ( Rply = yes,
      write('What is the full name of your file?'),nl,
      write('ABC import > '),
      readline([],Filename),
      consult(Filename),
      nl,nl,write('Import is complete. '),nl
    ),
    !,
    fail.

```



```

/*----- review_rules -----*/
/*
/* This command allows the viewing of rules in working memory. */

execute(review_rules) :-
    cls,
    nl,nl,nl,
    tab(26),write('ABC Rules in Working Memory'),nl,
    tab(26),write('====='),nl,nl,
    ( Rule : if Conds then Conclusion,
      write(Rule),write(' if'),nl,
      write_conditions(Conds),nl,
      write(' then '),write(C Conclusion),nl,nl,
      get0(_),fail
    ;
      write('No more rules can be found. '),nl
    ),
    !,
    fail.

/*----- review_frame -----*/
/*
/* This command allows the reviewing of frames in working memory. */
/* It prompts the user for the frame to be reviewed. */

execute(review_frame) :-
    nl,nl,
    write('What is the name of the frame you wish to review?'),
    nl,
    write('ABC frame review > '),
    readline([],Frame_Name),
    cls,
    nl,nl,nl,
    tab(17),
    write('ABC Frame Review - For the Frame: '),
    write(Frame_Name),
    nl,tab(15),
    write('====='),
    nl,nl,
    ( frame : Frame_Name slot Slots,
      write_frame(Slots)
    ;
      tab(20),
      write('Sorry, No frame exists with that name. '),
      nl,nl
    ),
    !,
    fail.

```

```

write_frame([]).
write_frame([Slot,Facet|Values]) :-
    nl,nl,tab(10),
    write('Slot = '),write(Slot),
    write(' and Facet = '),write(Facet),nl,
    tab(10),write('Values = '),writelist(Values).
write_frame([Slot,Facet|Values] slot Rest) :-
    nl,nl,tab(10),
    write('Slot = '),write(Slot),
    write(' and Facet = '),write(Facet),nl,
    tab(10),write('Values = '),writelist(Values),
    get0(_),
    write_frame(Rest).

/*----- add_goal -----*/
/*
/* This command allows on-line adding of goals to the front of the
/* goal list.
*/

execute(add_goal) :-
    nl,nl,
    write('Enter in the new goal in the form of an OAV triple.'),
    nl,
    write('(i.e.  graduation_class is Class '),
    nl,nl,
    write('ABC add_goal > '),
    readline([],Triple),          /* Goals are added without the normal */
    name(Triple,Tpl_List),        /* constraints associated with the   */
    convert_triple(Tpl_List,List), /* Prolog read predicate.           */
    condition(List,New_Goal),     /* Conditioning refers to writing the */
    get_goals(Goals),             /* goal out to a file after it has   */
    ( Goals = [],                 /* been converted to its new format. */
      assert(goals: New_Goal)     /* This is necessary in Prolog to    */
    ;                               /* make the Prolog database accept   */
      retract(goals : Goals),     /* variables.                         */
      assert(goals : New_Goal and Goals)
    ),
    !,
    fail.

```

```

/*----- condition -----*/
/*
/* The condition predicate takes triples which were read using ABC's */
/* readline predicate and converted from the quoted form to the list */
/* form and writes them out to a temporary file called "ABC.TMP". */
/* It then reads the file back into the Prolog database using the */
/* Prolog read predicate. This is necessary because words written */
/* beginning with an underscore or a capital letter is meant to be */
/* a variable but ABC's readline predicate cannot do this. The only */
/* way a word can be seen as a variable is if it meets the Prolog */
/* syntax requirements and is read into the Prolog database using */
/* the Prolog read predicate. */

condition(List,New_List) :-      /* The problem when converting a */
    tell('abc.tmp'),             /* quoted triple to a list is that */
    write(List),                 /* capitalized variables are not */
    write('.'),                 /* recognized as variables because */
    told,                       /* they are not read by the Prolog */
    see('abc.tmp'),             /* reader. condition/2 writes the */
    read(New_List),             /* the list out to a temporary file */
    seen.                       /* and reads it back so variables */
                                /* are seen as variables. */

get_goals(Goals) :-             /* get_goals/1 is used in lieu of */
    goals : Goals.             /* "goals : Goals" because it will */
get_goals([]).                 /* always succeed and it will return */
                                /* an empty list if no goals exist. */

/*----- add_frame -----*/
/*

execute(add_frame) :-
    cls,
    nl,nl,nl,
    tab(33), write('ABC Add_Frame'),nl,
    tab(25), write('====='),nl,nl,
    tab(25), write('Frame Name (or quit.) : '),
    readline([],FrameName),nl,nl,    /* Read in the name of the */
    ( FrameName = quit              /* frame along with its slot */
    ;                               /* name, facet type, and slot */
        tab(25),write('Slot Name : '), /* value.
        readline([],Slotname),nl,nl,
        tab(25),write('Facet Type: '),
        read_facet(Facet),nl,nl,
        tab(25),write('Slot Value: '), /* If you enter in a value for a */
        readline([],SlotValue),!,    /* slot with a "if-added" demon, */
        fput(FrameName,Slotname,Facet,SlotValue) /* then "fput" will */
    ),                               /* cause the demon to execute. */
    !,
    fail.

```

```

/*----- read_facet -----*/
/*
/* The read_facet predicate safeguards the user from entering in a
/* facet-type which is not recognizable by ABC.
/*
/* Constraints: read_facet(-).
*/

read_facet(Facet) :-
    readline([],Reply),
    ( member(Reply,[value,default,if_needed,if_added,if_removed]),
      Facet = Reply
    ;
      nl,write('You must enter a legal FACET (e.g. value, default,
if_needed, '),
      nl,write('if_added, if_removed). Please try again. '),
      nl,write('Facet type: '),
      read_facet(Facet)
    ).

/*----- delete_goal -----*/
/*

execute(delete_goal) :-
    nl,nl,
    write('Enter goal to be deleted. '),nl,
    write('NOTE: The goal must be in the form of a quoted triple. '),nl,nl,
    write('ABC Delete Goal > '),
    readline([],Qgoal),
    atom(Qgoal),
    !,
    name(Qgoal,Goal),
    convert_triple(Goal,Uncond_Goal), /* Convert the goal from a
condition(Uncond_Goal,Cond_Goal), /* quoted triple to a triple in
retract(goals : Goals),!, /* form of a list. The goal is
remove_goal(Cond_Goal,Goals,New_Goals), /* deleted by extracting
( New_Goals = [] /* the goal list, using the ABC
; /* predicate "remove_goal" to
assertz(goals : New_Goals) /* obtain a list of goals minus
),nl,nl, /* the one deleted, and assert
write('Deletion complete and successful. '),nl, /* this goal back
!, /* into the Prolog database.
fail.

/*----- remove_goal -----*/
/*
/* The remove_goal predicate removes the goal provided as its first
/* argument from the goal list provided as its second argument and
/* returns the modified goal list as its third argument. If the
/* last goal is removed, what is returned is a empty list.
/*
/* Constraints: remove_goal(+,+,-).
*/

```

```

remove_goal(Goal,Goal,[]).      /* Removing the only goal.      */
remove_goal(Goal,Goal and Addl_Goals,Addl_Goals). /* The first goal. */
remove_goal(Goal,Other_Goal and Addl_Goals,New_Goals) :-
    remove_goal(Goal,Addl_Goals,Returned_Goals),
    ( Returned_Goals = [],
      New_Goals = Other_Goal
    );
    New_Goals = Other_Goal and Returned_Goals
).
remove_goal(_,Goals,Goals).      /* Trying to remove a goal which does */
                                /* not exist.                          */

/*----- save_wm -----*/
/*                                                                */

execute(save_wm) :-
    save_wm,                    /* Call the "save_wm" predicate and go back to */
    !,                          /* the top level ABC prompt.                  */
    fail.

/*----- rules -----*/
/*                                                                */

execute(rules) :-
    cls,
    rules,                      /* Call the "rules" predicate and go back to */
    !,                          /* the top level ABC prompt.                  */
    fail.

```

```

/*----- syntax -----*/
/*
/* This command will assist the developer/user to determine where
/* syntax errors may exist in his knowledge base. It will write
/* his knowledge base one term at a time on the display screen.
/* When an error does appear, the user will notice the last terms
/* read successfully and be able to determine the whereabouts of his
/* error.
*/

execute(syntax) :-
    nl,nl,
    write('What knowledge base file would you like to check?'),
    nl,
    write('NOTE: Remember the file must have the extension of .kb.'),
    nl,nl,
    write('ABC Syntax Checker > '),
    readline([],Filename),
    !,
    name(Filename,ASCII_Name),
    check_ext(ASCII_Name,Ext),
    ( Ext = kb; Ext = abc),
    cls,
    nl,nl,nl,
    tab(30), write('ABC Syntax Checker'),nl,
    tab(10),
    write('====='),
    nl,nl,
    see(Filename),
    repeat,
    read(Term),
    ( Term = end_of_file,
      seen,
      nl,nl,
      tab(20), write('Reading of file '),
      write(Filename),
      write(' is complete.'),
      nl, tab(20), write('No errors were encountered'),nl
    ;
      Term = Rule : if Conds then Conclusion,
      tab(10), write(Rule),
      write(' is syntatically correct.'),nl,nl,
      fail
    ;
      tab(10), write(Term),
      write(' is syntatically correct.'),nl,nl,
      fail
    ),
    !,
    fail.

```

```

/*----- commands -----*/
/*
/* This ABC command simply displays all the available commands which
/* may be entered at the ABC command prompt. This is done as a
/* memory jogger for the user.
*/

execute(commands) :-
    commands,
    !,
    fail.

commands :-
    /* Displays all the available ABC commands on
    /* the current output device.
    cls,
    nl,nl,
    tab(16),
    write('ABC Command Line Commands Available to the User'),
    nl,
    tab(12),
    write('====='),
    nl,nl,
    repeat,
    tab(12), write('help          load          import'),nl,
    tab(12), write('consultation  trace        restore'),nl,
    tab(12), write('quit          commands     syntax'),nl,
    tab(12), write('rules        save_wm      review_frame'),
    nl,
    tab(12), write('review_goals  add_goal     delete_goal'),
    nl,
    tab(12), write('add_frame    add_slot     add_value'),nl,
    tab(12), write('delete_frame delete_slot   delete_value'),
    nl,nl.

/*----- any invalid command -----*/
/*
/* If the command the user types in is not recognized, then the fol-
/* lowing error message will be presented to the user.
*/

execute(_) :-
    put(7),          /* First thing that happens is an annoying beep */
    nl,nl,           /* then the user gets the following message.
    write('Bad ABC command. Consult User''s Manual or On-line help.'),
    nl,
    !,
    fail.

```

```

/*****
/*
/*                                     is_known/4                                     */
/*
/* The is_known predicate is the primary predicate which controls
/* the inference characteristics of ABC. The is_known predicate is
/* normally called upon by a rule to try to infer more implicit
/* knowledge from explicit facts and rules. It also returns the
/* certainty factor which the triple or structure is known.
/*
/* Constraints: is_known(+,+,?,-)
/* The structure of the first argument must be in the form of a
/* ABC triple or an ABC rule condition.

/* A conditional part of a rule is known if the conjunctive parts of
/* the conditions are both known. The CF will be the lesser of the
/* two.

is_known(Trpl_1 and Trpl_2, Rule_No, Cond_No, CF) :-
    nonvar(Rule_No),
    ( var(Cond_No), Cond_No = 1 ; true),
    is_known(Trpl_1, Rule_No, Cond_No, CF1),
    New_Cond_No is Cond_No + 1,
    is_known(Trpl_2, Rule_No, New_Cond_No, CF2),
    ( CF1 >= CF2, CF = CF2
    ;
      CF = CF1
    ).

/* A conditional part of a rule is known if either of the disjunc-
/* tive parts of the condition is known. If both are known, then
/* the two parts are treated as two separate rule conditions through
/* backtracking, with each CF giving support to the same conclusion.

is_known(Trpl_1 or Trpl_2, Rule_No, Cond_No, CF) :-
    nonvar(Rule_No),
    ( var(Cond_No), Cond_No = 1; true),
    ( is_known(Trpl_1, Rule_No, Cond_No, CF1),
      CF = CF1
    ;
      is_known(Trpl_2, Rule_No, Cond_No, CF2),
      CF = CF2
    ).

```



```

/* A conditional negated triple is known if the unnegated triple    */
/* is not known. The CF will always be 100.                          */

```

```

is_known(not [Obj,Attr,Val], Rule_No, Cond_No, 100) :-
    nonvar(Rule_No),
    ( var(Cond_No), Cond_No = 1; true),
    ( not is_known([Obj,Attr,Val], Rule_No, 666, CF),
      retract_temp_rules(Rule_No)
    );
    retract_temp_rules(Rule_No),
    CF < 10
),
assert_rule_trace(Rule_No, Cond_No, confirmed_not : [Obj,Attr,Val]).

```

```

/* The retract_temp_rules predicate retracts temporary assertions of */
/* rules which were placed into the trace mechanism when trying to   */
/* solve a negated (NOT) triple.                                       */
/* Constraints: retract_temp_rules(+).                                  */

```

```

retract_temp_rules(Rule_No) :-
    rule_trace(Rule_No, Cond_No, _),
    ( Cond_No >= 666,
      retract(rule_trace(Rule_No, Cond_No, _))
    );
    true
), fail.
retract_temp_rules(_) :- !.

```

```

/*                               is_known method 1                      */
/* If a triple has already been determined to fail, either by being  */
/* denied by the user, by being confirmed not to be true, or as      */
/* being a triple concluded by some method to have a certainty       */
/* factor of zero, then "is_known(Triple)" fails without further    */
/* searching.                                                         */

```

```

is_known([Obj,Attr,Val|_], Rule_No, Cond_No, _) :-
    ( denied([Obj,Attr,Val])
    ; trace(confirmed_not : [Obj,Attr,Val|_])
    ; trace(_ : [Obj,Attr,Val,0])
    ; trace(_ : [Obj,Attr,Val,0] derived_from _)
    ),
    !, fail.

```

```

/*                                     is_known method 2                                */
/* If a triple has already been determined to succeed, either by                     */
/* being confirmed by the user or by its presence in the trace, then                 */
/* is_known(Triple) will succeed and the search for additional ways                 */
/* to find out if the triple is known will be discontinued.                         */

```

```

is_known([Obj,Attr,Val|_], Rule_No, Cond_No, CF) :-
    search_trace([Obj,Attr,Val],CF),
    assert_rule_trace(Rule_No, Cond_No, [Obj,Attr,Val,CF]).

```

```

/* The search_trace predicate searches both the main trace and the                 */
/* rule trace to determine if a triple is present in either of the                 */
/* traces. If it is, the predicate will succeed with some CF.                     */
/* Constraints: search_trace(+,-).                                                  */

```

```

search_trace([Obj,Attr,Val],CF) :-
    ( confirmed([Obj,Attr,Val|_]),
      CF = 100
    ;
      trace(_ : [Obj,Attr,Val,100]),
      CF = 100
    ;
      ( trace(_ : [Obj,Attr,Val|CF1] derived_from _)
        ;
          rule_trace(_,_,[Obj,Attr,Val|CF1])
        ),
      ( CF1 = [], CF = 100
        ; CF1 = [CF], CF = 100
      )
    ), !.
search_trace(_,_) :- !, fail.

```

```

/*                                     is_known method 3                                */
/* If a triple [Obj,Attr,Val] is a valid Prolog fact or goal,                       */
/* Attr(Obj,Val), and this Prolog term can succeed, then "is_known                 */
/* (Triple)" will also succeed and the search for another way to                   */
/* find is_known(Triple) is discontinued.                                          */

```

```

is_known([Obj,Attr,Val], Rule_No, Cond_No, 100) :-
    not(Attr == is),
    T =.. [Attr,Obj,Val],
    T,                                     /* Prolog goal Attr(Obj,Val). */
    assert_rule_trace(Rule_No, Cond_No, solved : [Obj,Attr,Val,100]),
    !.

```

```

is_known([Obj,=,Val], Rule_No, Cond_No, 100) :-
    Obj is Val,
    assert_rule_trace(Rule_No, Cond_No, solved : [Obj,Attr,Val,100]),
    !.

```

```

/*                      is_known method 4                      */
/* If a triple can be found in a frame, assuming all knowledge in */
/* frames to have a certainty factor of 100, then is_known(Triple) */
/* will succeed. */

```

```

is_known([Obj,Attr,Val], Rule_No, Cond_No, 100) :-
    frame : Obj slot Slots,
    frame_get(Obj,Attr,Val_List),
    member(Val,Val_List),
    assert_rule_trace(Rule_No, Cond_No, frame_fact : [Obj,Attr,Val,100]).

```

```

/*                      is_known method 5                      */
/* If the triple is a fact, and the fact is known with a CF of 100, */
/* then further search is discontinued. Otherwise, if a fact can be */
/* found with a CF less than 100, this predicate will return this CF */
/* and the search for additional paths will continue. */

```

```

is_known([Obj,Attr,Val|_], Rule_No, Cond_No, CF) :-
    fact : [Obj,Attr,Val|CF1],
    ( ( CF1 = [] ; CF1 = [cf,100]), CF = 100, !
    ;
        CF1 = [cf,CF]
    ),
    assert_rule_trace(Rule_No, Cond_No, fact : [Obj,Attr,Val,CF]).

```

```

/*                      is_known method 6                      */
/* A triple is known if it is the head of a rule and the conditions */
/* of the rule are known. If the rule certainty factor is equal to */
/* 100 and all the conditions are known with a certainty factor of */
/* 100 then the is_known(Triple) succeeds with a certainty factor of */
/* 100. */

```

```

is_known([Obj,Attr,Val], Rule_No, Slot_No, CF) :-
    Rule : if Conds then [Obj,Attr,Val|Rule_CF],
    retract_all(rule_trace(Rule,_,_)),
    clean_up_why_trace(Obj,Attr,Val),
    asserta(why_trace(Rule : [Obj,Attr,Val|Rule_CF] derived_from Conds)),
    is_known(Conds, Rule, 1, Conds_CF),
    ( ( Rule_CF = []
      ; Rule_CF = [cf,100]
    ),
        CF = Conds_CF
    ;
        Rule_CF = [cf,R1_CF],
        CF is Conds_CF * R1_CF / 100
    ),
    assert_rule_trace(Rule_No, Slot_No, Rule : [Obj,Attr,Val,CF]
        derived_from Conds).

```

```

/*                                     is_known method 8                               */
/* A triple is known if (a) the rule-base classifies it as "askable" */
/* and if (b) the user confirms it. All explicitly declared ask- */
/* ables in the rule-base are in the form of:                        */
/* "askable : 'Obj Attr Val' derived from Question and Answers" */
/* where Question is the question that will be asked, and Answers */
/* are a list of valid answers. An askable will only be asked if */
/* there is currently not a Obj-Attr-? triple already in the trace. */
/* If the askable is answered, the triple goes into the trace and */
/* is_known(Triple) succeeds only if the selected answer matches the */
/* value of Val. In either case, is_known(Triple) is not resatis- */
/* fiable.                                                            */

```

```

is_known([Obj,Attr,Val], Rule_No, Cond_No, 100) :-
    not_in_trace(Obj,Attr),
    askable : [Obj,Attr,Val] derived from Question and Valid Answers,
    ask_question([Obj,Attr,Val],Question,Valid Answers,Value,CF),
    assert_in_trace(askable : [Obj,Attr,Value,CF]),
    assert_rule_trace(Rule_No, Cond_No, [Obj,Attr,Value,CF]),
    Value = Val,
    CF > 10, !.

```

```

/*----- assert_rule_trace -----*/
/*                                     */
/* The assert_rule_trace predicate stores a temporary trace in the */
/* Prolog database. If the rule succeeds in which this temporary */
/* trace is associated with, then the temporary trace is also as- */
/* serted into the main why trace. Each of these temporary traces */
/* are distinguished from other temporary traces by using the rule */
/* number and the condition number. */
/*                                     */
/* Constraints: assert_rule_trace(+,+,+). */

```

```

assert_rule_trace(Rule_No, Cond_No, Known_Trpl) :-
    nonvar(Rule_No),
    nonvar(Cond_No),
    nonvar(Known_Trpl),
    retract_all(rule_trace(Rule_No, Cond_No, _)),
    assertz(rule_trace(Rule_No, Cond_No, Known_Trpl)),
    !.

```

```

/*----- not_in_trace -----*/
/*                                     */
/* The not_in_trace predicate checks both the main trace and the */
/* temporary rule traces to see if a fact or rule has already been */
/* been asserted with the same object and attribute pair. If there */
/* are no assertions found, the predicate will return successful, */
/* otherwise, it will fail. */
/*                                     */
/* Constraints: not_in_trace(+,+). */
/* Both arguments must be atoms. */

```

```

not_in_trace(Obj,Attr) :-
    not
    ( trace(_ : [Obj,Attr|_])
    ;
      trace(_ : [Obj,Attr|_] derived_from _)
    ;
      rule_trace(_,_,_ : [Obj,Attr|_])
    ;
      rule_trace(_,_,_ : [Obj,Attr|_] derived_from _)
    ), !.

/*----- clean_up_why_trace -----*/
/*
/* The clean_up_why_trace predicate is necessary because when rules
/* and their conditions are placed into the why trace, they are not
/* removed if the rule fails. In such an instance, the why trace
/* gets cluttered. This predicate increases the efficiency of the
/* why trace by removing unnecessary assertions. It does this by
/* requiring assertions to either be part of the goal itself, or
/* part of the rule's condition which is just above it in the order
/* of when it was asserted.
/*
/* Constraints: clean_up_why_trace(+,+,+).
/* Arguments must be atoms.
*/

clean_up_why_trace(Obj,Attr,Val) :-
    why_trace(Top_Why),
    ( Top_Why = goal : _, !
    ;
      Top_Why = Rule : Triple derived from Conds,
      part_of(Obj, Attr, Val, Conds), !
    ;
      retract(why_trace(_)),
      clean_up_why_trace(Obj,Attr,Val)
    ), !.
clean_up_why_trace(Obj,Attr,Val) :- !.

part_of(Obj,Attr,Val, Conds) :-
    ( Conds = [Obj,Attr,Val|_], !
    ;
      Conds = not [Obj,Attr,Val|_], !
    ;
      Conds = [Obj,Attr,Val|_] and Rest, !
    ;
      Conds = not [Obj,Attr,Val] and Rest, !
    ;
      Conds = [Obj,Attr,Val|_] or Rest, !
    ;
      Conds = not [Obj,Attr,Val|_] or Rest, !
    ;

```

```

        Conds = Triple and Rest, !,
        part_of(Obj,Attr,Val,Rest), !
    ;
        Conds = Triple or Rest, !,
        part_of(Obj,Attr,Val,Rest), !
    ;
        !, fail
    ), !.

/*----- explanation facility -----*/

/* The explain_why predicate explains why a question is being asked */
/* to the user if he requests an explanation. It uses the ABC pred- */
/* icate "justify" to assist in this utility. */
/* Constraints: explain_why(+). */

explain_why(Triple) :-
    retract(why_trace(Why_trace)),!,
    nl,
    justify(Triple,Why_trace),
    ( Why_trace = goal : _,
      true
    ;
      Why_trace = Rule : Conclusion derived from Conditions,
      nl,nl,write('Do you wish to continue trace? (y/n) : '),
      get_reply(Rply),
      ( Rply = yes,
        explain_why(Conclusion)
      ;
        true
      )
    ;
      true
    ),
    nl,
    asserta(why_trace(Why_trace)),!.

/* If the why trace is empty and a question is being prompted to the */
/* user for which he would like an explanation, then the question */
/* must be from an initial-askable and the only explanation that can */
/* be given is shown below. */

explain_why(_) :-
    nl,tab(2),
    write('The answer to this question is needed to solve the main goal.'),
    nl, !.

```

```

/* The justify predicate justifies the goals and rules found in the */
/* why trace by pretty printing them out to the display screen.    */
/*                                                                    */
/* Constraints: justify(+,+).                                         */

```

```

justify(Goal,goal : Goal) :-
    nl,write('This will satisfy the goal '),
    writelist(Goal),
    write('.').
justify(Triple,Rule : Conclusion derived from Conditions) :-
    nl,write('I can use '),writelist(Triple),
    write(' to help satisfy '),write(Rule),nl,
    write('which if satisfied, will give me '),
    writelist(Conclusion), write('.').

```

```

/*----- Frame Representation -----*/
/* This section is an expanded version of the frame-based knowledge */
/* representation described by Cuadrado, J.L. & C.Y. in "AI in Compu- */
/* ter Vision," BYTE, volume 11, nbr 1, January 1986, pp. 237-258.  */
/*                                                                    */
/* Possible Facets: value                                             */
/*                  default                                           */
/*                  if_needed                                         */
/*                  if_added                                           */
/*                  if_removed                                         */
/*                                                                    */
/*-----*/

```

```

/* The frame_get predicate retrieves a list of values from a frame. */
/* It goes through a predetermined search to search for its values.  */
/*                                                                    */
/* Constraints: frame_get(+,+,-).                                     */

```

```

frame_get(Frame,Slot,Value_List) :-          /* First, look for a value */
    fget(Frame,Slot,value,Value_List),!.     /* under the "value" slot */
frame_get(Frame,Slot,Value_List) :-          /* attribute, if not found, */
    fget(Frame,Slot,default,Value_List),!.   /* look at "default", "if- */
frame_get(Frame,Slot,Value_List) :-          /* needed", and "ako" slots */
    fget(Frame,Slot,if_needed,[Demon]),      /* in that order.          */
    !,
    F =.. [Demon,Frame,Value_List],
    F,!.
frame_get(Frame,Slot,Value_List) :-          /* Allows for multiple    */
    fget(Frame,ako,value,Parent_List),       /* inheritance.           */
    member(Parent,Parent_List),
    assertz(temp_trace(via_ako : [Frame,Parent])),
    frame_get(Parent,Slot,Value_List),!.

```

```

/* The frame_put predicate puts a value into a frame. It first sees */
/* if there is a "if_added" demon associated with the frame and */
/* executes the demon procedure if it is. Otherwise, it places the */
/* value into the frame and slot specified under the slot-attribute */
/* of value. If the slot or frame doesn't exist, they are created. */
/* */
/* Constraints: frame_put(+,+,+). */

```

```

frame_put(Frame,Slot,Value) :-
    get_rule(Frame,Slot,if_added,Rule),
    !,
    fput(Frame,Slot,value,Value),
    F =.. [Rule,Frame,Value],
    F.

```

```

frame_put(Frame,Slot,Value) :-
    fput(Frame,Slot,value,Value).

```

```

/* The frame_remove predicate removes all the values for a partic- */
/* ular frame and slot. This includes all slot-attributes. This */
/* deletion effectively deletes the slot. */
/* */
/* Constraints: frame_remove(+,+). */

```

```

frame_remove(Frame,Slot) :-
    get_rule(Frame,Slot,if_removed,Rule),
    !,
    F =.. [Rule,Frame],
    F,
    repeat,
    not fdelete(Frame,Slot,_,_).

```

```

frame_remove(Frame,Slot) :-
    repeat,
    not fdelete(Frame,Slot,_,_).

```

```

/* The fget predicate is the utility predicate which actually does */
/* the work of the "frame_get" predicate. */
/* */
/* Constraints: fget(+,+,+,-). */

```

```

fget(Frame,Slot,Facet,Value_List) :-
    frame : Frame slot Slots,
    sget(Slots,Slot,Facet,Value_List), !.

```



```

/* The sget predicate is a recursive predicate which obtains the */
/* the values of the given frame/slot/slot-facet combination. */
/* */
/* Constraints: sget(+,+,+,-). */

sget(Slots,Slot,Facet,Value_List) :-
  ( Slots = [Slot,Facet|Value_List], !
  ;
    Slots = [Slot,Facet|Value_List] slot Rest_1, !
  ;
    Slots = Slots_1 slot Slots_2, !,
    sget(Slots_2,Slot,Facet,Value_List)
  ), !.

/* The fput predicate is the utility predicate which does the work */
/* for the frame_put predicate. Given a frame-name, a slot-name, */
/* a slot-facet, and a value, the fput predicate will place the */
/* value into that particular frame/slot/attribute combination if it */
/* exists, or create it if it doesn't exist. */
/* */
/* Constraints: fput(+,+,+,+). */

fput(Frame,Slot,Facet,Value) :-
  frame : Frame slot Slots,!,
  add_value(Slot,Facet,Value,Slots,New_Slots),
  retract(frame : Frame slot Slots),
  assertz(frame : Frame slot New_Slots), !.

fput(Frame,Slot,Facet,Value) :-
  assertz(frame : Frame slot [Slot,Facet,Value]).

/* The add_value predicate adds a value to a slot when the frame */
/* is known to exist. If either the slot or the slot attribute is */
/* not present, then they will be created in order to hold the new */
/* value which is being added. */
/* */
/* Constraints: add_value(+,+,+,+,-). */

add_value(Slot,Facet,Value,Slots,New_Slots) :-
  ( Slots = ([Slot,Facet|Values]),
    New_Slots = ([Slot,Facet,Value|Values])
  ;
    Slots = ([Slot,Facet|Values] slot Rest),
    New_Slots = ([Slot,Facet,Value|Values] slot Rest)
  ;
    Slots = (Slot_1 slot Slot_2),
    add_value(Slot,Facet,Value,Slot_2,New_Slots_2),
    New_Slots = (Slot_1 slot New_Slots_2)
  ;
    New_Slots = ([Slot,Facet,Value] slot Slots)
  ).

```

```

/* The fdelete predicate will delete a particular value from a frame */
/* if the frame-name, slot-name, slot-facet, and value are provided. */
/*                                     */
/* Constraints: fdelete(+,+,+,+). */

```

```

fdelete(Frame,Slot,Facet,Value) :-
    frame : Frame slot Slots,
    delete_value(Slot,Facet,Value,Slots,New_Slots),
    ( New_Slots = [],
      retract(frame : Frame slot Slots)
    ;
      retract(frame : Frame slot Slots),
      assertz(frame : Frame slot New_Slots)
    ).

```

```

/* The delete_value predicate will delete a value from a slot and */
/* return a new list of slots if given the slot-name, slot-facet and */
/* the value which needs to be deleted along with the original list */
/* of slots. */
/*                                     */
/* Constraints: delete_value(+,+,+,+,-). */

```

```

delete_value(Slot,Facet,Value,Slots,New_Slots) :-
    ( Slots = ([Slot,Facet,Value]),!,
      New_Slots = []
    ;
      Slots = ([Slot,Facet|Values]),!,
      remove(Value,Values,New_Values),
      New_Slots = ([Slot,Facet|New_Values])
    ;
      Slots = ([Slot,Facet,Value] slot Rest1),!,
      New_Slots = (Rest1)
    ;
      Slots = ([Slot,Facet|Values2] slot Rest2),!,
      remove(Value,Values2,New_Values2),
      New_Slots = ([Slot,Facet|New_Values2] slot Rest2)
    ;
      Slots = (List slot Rest3),
      delete_value(Slot,Facet,Value,Rest3,New_Rest3),!,
      ( New_Rest3 = [],
        New_Slots = List
      ;
        New_Slots = (List slot New_Rest3)
      )
    ).

```

```

/* The get_rule predicate will retrieve a demon procedure (rule) */
/* from the frame structure either from the Frame and Slot specified */
/* or from a parent/grandparent of the Frame specified. */
/* */
/* Constraints: get_rule(+,+,+,-). */
/* The third argument must be one of the demon attributes used in */
/* ABC, i.e. if_needed, if_added, etc. */

```

```

get_rule(Frame,Slot,Type,Rule) :-
    fget(Frame,Slot,Type,Rule).
get_rule(Frame,Slot,Type,Rule) :-
    fget(Frame,ako,value,Parent),
    get_rule(Parent,Slot,Type,Rule).

```

```

/* The print_goals predicate prints out the goal's list on the dis- */
/* play monitor with one goal per line. It is used within ABC to */
/* display the goals during a trace. This predicate may be used to */
/* print out any number of OAV triples, which are in list form, */
/* seperated by the ABC defined operator "and". */
/* */
/* Constraints: print_goals(+). */

```

```

print_goals([]).
print_goals(Goals) :-
    ( Goals = [Obj,Attr,Val] and Rest
    ;
      Goals = [Obj,Attr,Val],
      Rest = []
    ),
    tab(23),
    write(Obj),
    tab(3),
    write(Attr),
    tab(3),
    write(Val),
    nl,
    print_goals(Rest).

```

```

/* The write_conditions predicate displays the conditions of a rule */
/* in such a way to make it easy to read during a trace. Its a */
/* "pretty printer" of sorts. It will print out any condition which */
/* ABC is capable of understanding. */
/* */
/* Constraints: write_conditions(+). */

```

```

write_conditions([X,Y,Z]) :-
    tab(8),
    writelist([X,Y,Z]),
    nl.

```

```

write_conditions(not [X,Y,Z]) :-
    tab(4),
    write('Not '),
    writelist([X,Y,Z]),
    nl.
write_conditions([X,Y,Z] and Conditions) :-
    tab(8),
    writelist([X,Y,Z]),
    write(' and'),
    nl,
    write_conditions(Conditions).
write_conditions(not [X,Y,Z] and Conditions) :-
    tab(4),
    write('Not '),
    writelist([X,Y,Z]),
    write(' and'),
    nl,
    write_conditions(Conditions).
write_conditions([X,Y,Z] or Conditions) :-
    !,
    tab(8),
    writelist([X,Y,Z]),
    write(' or'),
    nl,
    write_conditions(Conditions).
write_conditions(Conditions1 or Conditions2) :-
    write_conditions(Conditions1),
    tab(8),
    write('or'),
    nl,
    write_conditions(Conditions2).
write_conditions(not [X,Y,Z] or Conditions) :-
    tab(4),
    write('Not '),
    writelist([X,Y,Z]),
    write(' or'),
    nl,
    write_conditions(Conditions).

/* The convert_to_list predicate converts an atom to a list of words */
/* which the atom originally had separated by spaces. In ABC, this */
/* predicate was used to convert a quoted triple into a triple in */
/* which the object, attribute, and value were in a list form. */
/* */
/* Constraints: convert_to_triple(+,-). */

convert_to_list(Atom Triple, List_Triple) :-
    atomic(Atom Triple),
    name(Atom Triple, ASCII_List),
    tokenize(ASCII_List,[],List_Triple).

```

```

/* The tokenize predicate was derived from Claudia Marcus's book, */
/* "Prolog Programming", pp 205-209. This predicate will take a */
/* list of ASCII numbers, and convert them to a list of words. A */
/* word will be defined as anything which can be found before, */
/* between, or after an ASCII 32 (a space) and which can be con- */
/* verted from its ASCII form back to an atom. */
/* Constraints: tokenize(+,+,-). */

tokenize([],List,List).
tokenize([32|T],List,L) :-
    !,
    tokenize(T,List,L).
tokenize([H|T], List, L) :-
    get_rest_word(T,[H],Word,Rem),
    append(List,[Word],NewList),
    tokenize(Rem,NewList,L).

/* The get_rest_word predicate is used in conjunction with the */
/* tokenize predicate. The get_rest_word predicate, once the first */
/* letter of a word is seen, will go through the ASCII list append- */
/* ing ASCII numbers to the tail of a temporary list until it sees */
/* an ASCII 32 (a space), or it runs out of ASCII numbers. In */
/* either case, it converts the temporary list of ASCII numbers to */
/* an atom and returns this atom as the word. */
/* Constraints: get_rest_word(+,+,-,-). */

get_rest_word([],List,Word,[]) :-
    !,
    name(Word,List).
get_rest_word([32|T],List,Word,T) :-
    name(Word,List),
    !.
get_rest_word([H|T],List,Word,X) :-
    append(List,[H],NList),
    get_rest_word(T,NList,Word,X).

```

```

/* The get_reply predicate gets the next character placed into the */
/* current input device and determines whether its a carriage return,*/
/* a "y", a "n", a "w", or something else. If next character is */
/* either a carriage return or a "y", then get_reply will return the */
/* atom "yes". If the character is a "n", it will return "no". If */
/* the character is a "w", it will return "why". If the character */
/* is anything else, it will prompt the user to reenter the char- */
/* acter. */
/* */
/* Constraints: get_reply(-). */

```

```

get_reply(Reply) :-
    get0(User_Reply),nl,
    ( User_Reply = 13, Reply = yes,!           /* Carriage Return */
    ; User_Reply = 121, Reply = yes,!
    ; User_Reply = 110, Reply = no, !
    ; User_Reply = 119, Reply = why,!
    ;
    nl,nl,
    write('You must enter either a "y" or a "n", or if you wish, '),
    nl,write('just hit a return for yes. '),
    get_reply(Reply) ),!.

```

```

/*----- readline -----*/
/* */
/* This ABC predicate allows the user to respond to a prompt with- */
/* out the requirement that the user's response be a proper Prolog */
/* term. It will allow letters and numbers and a few symbols to be */
/* entered in via the keyboard. It will also allow the deletion of */
/* the last character using the backspace-rubout key. */
/* */
/* Constraints: readline(+,-). */

```

```

readline(Temp_List,Line) :-
    get0(Char),!,
    ( Char = 13,!,nl,           /* If the return key is pressed, */
      reverse(Temp_List,Line_List), /* the ASCII list is reversed */
      name(Line,Line_List)      /* and the line is generated. */
    ;
      Char = 8,!,              /* If the rubout key is pressed, */
      put(32),put(8),          /* we go back a space, put in a */
      Temp_List = [Head|Tail], /* blank, go back again and then */
      readline(Tail,Line)      /* go on with our corrected list.*/
    ;
      identifier(Char),!,      /* We place all identifiers in */
      readline([Char|Temp_List],Line) /* ASCII list and continue. */
    ;
      Char = 32,!,             /* We place blank spaces into */
      readline([32|Temp_List],Line) /* ASCII list and continue. */
    ;
      readline(Temp_List,Line),! /* All other inputs are ignored */
    ), !.                      /* but we continue to read. */

```

/*----- Utilities -----*/

```
reverse([],[]).
reverse([X|L],M) :-
    reverse(L,N),
    append(N,[X],M).
```

```
remove(_,[],[]).
remove(X,[X|L],M) :-
    !,
    remove(X,L,M).
remove(X,[Y|L],[Y|M]) :-
    remove(X,L,M).
```

```
writelist([]).
writelist([X|L]) :-
    write(X),
    tab(1),
    writelist(L).
```

```
member(X, [X|_]).
member(X, [_|Y]) :-
    member(X,Y).
```

```
retract_all(X) :-
    retract(X),
    fail.
retract_all(_) :- !.
```

```
append([],List,List).
append([X|L],M,[X|N]) :-
    append(L,M,N).
```

```
/* The function of the cls predicate is to clear the screen (CRT). */
/* The cls predicate is not used in Arity Prolog because it is pre- */
/* defined there. However, in Prolog-1 or other Prologs where it is */
/* it not predefined, the following predicate will suffice. */
/*
```

```
cls :-
    nl,nl,nl,nl,nl,nl,nl,nl,nl,nl,!.
*/
```

USER'S MANUAL - Appendix C
The AFIT Backward Chainer (ABC)
Expert System Shell

Introduction
=====

The ABC expert system shell (ESS) was designed to assist students pursuing the Artificial Intelligence (AI) sequence at AFIT. The Prolog code which makes up ABC is entirely written in standard Prolog, sometimes referred to as the Clocksin and Mellish (C&M) dialect. For the Prolog student, this means that the predicates within ABC can be studied or used in other similar programs. To the students who are mainly interested in building expert systems, this ESS provides several features which are usually found in commercial ESSs. Thus to these students, ABC can prove to be a good source of study to see how AI techniques and data structures are used in an ESS.

ABC uses the built-in backward chaining, top-down control-mechanism of the Prolog interpreter, via the ABC predicate `is_known/1`, to provide its inference engine. ABC has two basic types of knowledge representations: production rules and frames. Production rules allow "rules of thumb" to be utilized to capture the expertise, while frames allow the efficient and natural data structure necessary for many types of domains in which commercial ESSs are commonly used.

This user's manual consists of six parts. The first part is this introduction. The second part explains how to get into and out of ABC. It also explains a couple of points about installing and/or configuring

the ABC ESS. The third part shows all of the commands found in ABC, usually followed with an example of how they can be used. The commands are listed in alphabetical order to aid the user as a reference. The fourth part explains the knowledge structures of ABC and gives guidance on creating a knowledge base. The fifth part is for those users wishing to use demons in their ABC frames. The last section of this manual will step the beginning user through a wine advisory consultation.

Starting and Exiting ABC

Before You Start ABC - Since ABC is written in Prolog, the Prolog interpreter has to be loaded prior to using ABC. Consult your Prolog Interpreter's user's manual to determine how to load and consult Prolog. However, before loading the Prolog interpreter there is something you must consider. Special care was taken to write the code for ABC so that it would be portable to all computer environments having a C&M Prolog interpreter available. Most Prologs are a superset of C&M Prolog and thus some of the predicates defined in ABC may be included as built-in predicates of the Prolog interpreter you are using. If this is the case, you will need to "comment out" the ABC code which defines these predicates prior to using ABC or error warnings may appear.

There is no standardization in Prolog operator's precedence. Yet there exists two paradigms which are incompatible with each other. Most Prolog interpreters subscribe to one or the other of these operator precedence paradigms. Figures 1A and 1B show the ABC defined operator precedences and how they are defined in Arity Prolog and Prolog-1, respectively. Your disk should contain the Prolog source code for both Arity and Prolog1. If you are not using one of these three interpreters, consult your Prolog manufacturer's user's manual to see which one of these two paradigms are used and set your operator's precedence levels accordingly. If you have problems, or you get operator error messages, you will need to obtain assistance from your Prolog manufacturer's service center.

```

:- op(990, xfx, =).
:- op(980, xfy, :).
:- op(975, xfx, then).
:- op(970, xfy, slot).
:- op(970, fx, if).
:- op(965, xfx, derived_from).
:- op(960, xfy, or).
:- op(955, xfy, and).

```

Figure 1A: ABC Operator Definitions in Arity Prolog

```

?- op(250, xfx, :).
?- op(245, xfx, then).
?- op(240, xfy, slot).
?- op(240, fx, if).
?- op(235, xfx, derived_from).
?- op(230, xfy, or).
?- op(225, xfy, and).

```

Figure 1B: ABC Operator Definitions in Prolog-1

How to Start and Exit ABC - After the above steps have been completed and your Prolog interpreter is awaiting your input, use Prolog's built-in consult predicate to consult ABC. Most Prologs default to some predefined extension if one is not provided, so be careful and provide the extension if you are unsure about your Prolog interpreter. A typical consultation for using ABC on the Arity Prolog interpreter on the Z-248s (or IBM AT compatible) is shown below.

```
consult('abc.ari'). or consult(abc).
```

The Prolog consult commands listed above assumes that the ABC code has the extension of ".ari" and that this file exists on the current drive. Your input command to consult ABC may be quite different.

Once ABC is consulted, it will start up automatically providing you with an introduction screen. This screen will remind you to always use lower-case letters for commands and replies. A key point to observe here is that once you are in ABC, the prompt will change to "ABC >." If you don't see this prompt then you are not in ABC and you may need to get some assistance.

At the ABC prompt, you may type in one of several ABC commands. All of the commands will be explained in detail later in this user's manual. If you have a problem remembering commands, it is a good idea to remember at least the ABC command "commands." The "commands" command will provide a listing of all the commands available to you in the ABC ESS. Another helpful hint to new users is the command "help." The "help" command will provide some assistance to new users on a few of the more basic ABC commands.

If for some reason you are forced to leave ABC and go back into the underlying Prolog interpreter, you should always be able to re-enter ABC by typing the command "start." Usually, most information such as your rules, facts, frames, and assertions should be retained by the underlying Prolog interpreter. You can check this out with any of the "review" commands available in ABC.

Although ABC activates itself once you consult it into working memory, it does not bring with it any domain knowledge. You must provide the domain knowledge in the form of files which have frames and/or rules and/or facts via the use of one of the two ABC "load" commands. You may load more than one file providing the rules and/or data don't conflict. A second method of entering knowledge into working memory is through the

use of one or more of ABC's "add" commands, but this is not recommended except for adding small amounts of knowledge for testing.

Once you have entered the domain knowledge, you must provide ABC with one or more goals. You may load goals into working memory by including them in your knowledge base and have them loaded along with your knowledge, or you may load goals by using the ABC "add_goal" command.

After your knowledge base and goal(s) are loaded into working memory, you are then ready to start a consultation with ABC to resolve your goals. You may initiate a consultation with ABC by either the "go" or "consultation" commands. They are both identical. ABC will search your knowledge base during a consultation, asserting and deleting data and structures in order to solve your goal(s). If no solution can be found, a message to that effect will be presented.

To leave ABC naturally and go back to the Prolog interpreter, you must use the ABC "quit" command.

ABC Commands

All ABC commands are typed at the ABC command prompt. No parameters are ever passed with the ABC commands. The following is a complete description of all the ABC commands in alphabetical order.

add_frame

The add_frame command will add frames to the knowledge base via the keyboard. This is not recommended to enter in large number of frames in this manner. The ASCII file which makes up your knowledge base should be edited for large number of frame additions.

If you wish to add a either a frame-slot or a slot-value to a pre-existing frame, this command can also be used in this fashion. ABC will not erase a frame in favor of a newer addition; it will always append slots and slot-values when frames already exist with the correct name. A frame can be added in a manner similar to the example shown below.

ABC> add_frame

ABC add_frame

=====

Frame Name (or quit) : afit_student

Slot Name : class

Facet Type : default

Slot Value : gce

Example of Adding a Frame Using the add_frame Command.

The frame name is generally an object and the slot name is usually the object's attribute. The slot value is usually the value associated with the slot name. If more than one exist, the add_frame procedure must be accomplished for each value. The facet type must be either "value", "default", "if_needed", "if_added", or "if_removed". If either of the last three facet types are used, the slot value will then become the name of the associated demon procedure. Refer to part five of this user's manual for more information on demon procedures.

add_goal

The add_goal command allows the user to enter a goal into the knowledge base. Adding a new goal to the knowledge base does not delete any goals which may have pre-existed. Using the add_goal command appends a new goal to the front of the goals list. If the goal must be placed somewhere other than at the front, a combination of delete_goal and add_goal commands must be performed. An example of adding a goal to the knowledge base is shown below.

```
ABC> add_goal
```

```
Enter in the new goal in the form of an OAV triple.  
(i.e.  graduation_class is Class)
```

```
ABC add_goal > recommended_wine is Wine
```

Example of Adding a Goal Into The Knowledge Base

Notice that the OAV triple is entered with only one or more spaces separating the individual components of the triple. Do not attempt to use a carriage return between the individual components of the triple. A carriage return identifies the end of an add_goal entry.

commands

The command "commands" will present a listing of all of the current ABC commands which are available to the user.

consultation

The command consultation will start a new consultation. The command "go" is synonymous with the command consultation.

delete_goal

The command delete_goal will delete a goal from the goal list regardless of its relative location in the goal list. The goal you wish to delete must be an OAV triple. An example of goal being deleted is shown below.

```
ABC > delete_goal
```

```
Enter goal to be deleted.  
(i.e.   recommended_vine is Vine)
```

```
ABC delete_goal > tweetie owner Owner
```

Example of Deleting a Goal Using The delete_goal Command

If you try to delete a goal that does not exist, ABC will continue to its main prompt and ignore your request for deleting the goal.

go

The go command is the functionally equivalent to the consultation command. Since it is shorter to type in, most users may prefer to use this command as opposed to the consultation command.

help

The help command gives basic information about the five commands which beginning users would normally use most often. The five commands are: "load", "remove_kb", "consultation", "restore", and "quit".

import

The import command consults Prolog code directly into the Prolog database without going through any intermediate steps. An example of this command is shown below.

```
ABC > import
```

```
This ABC command imports Prolog code into the Prolog database.  
Do you wish to continue? (<y>, n): y
```

```
What is the full name of your file?  
ABC import > a:pets.aux
```

Example of The import Command

load

The load command is used to load a knowledge base from a file to the Prolog database. There are two types of knowledge bases: the one with user-friendly quoted triples and the one with triples structured in Prolog lists. The user-friendly knowledge base file should have the

extension of ".kb". When it is first loaded, it is parsed in order to convert all of its quoted triples into Prolog lists. This conversion process creates a second file with the same prefix as the ".kb" file, yet with the extension of ".abc". Since ".abc" files will load much faster than the ".kb" files, always load the ".abc" file if a current copy exists.

An example of loading a knowledge base file is shown below.

Enter the name of the file where your knowledge base is stored, or enter <Return> to abort.

Filename, including path is: c:\abc\kb\wine

Example of Loading a Knowledge Base File

Notice in the example that if you wish to abort the load operation after executing it, just press the Return or Enter key.

quit

The quit command is the only natural way to terminate ABC.

remove_kb

The remove_kb command removes a knowledge base from the Prolog database. It does this by removing all of the facts, frames, and rules. This command will give adequate warning and will allow you to abort the operation if you deem necessary. This command does not remove any demon procedures or other user-defined Prolog predicates which may have been loaded in via an auxiliary file.

restore

The restore command removes any assertions made in previous consultations from the Prolog database. This command differs from the remove_kb command because it does not remove any of the facts, rules, or frames. If you have made mistakes during a consultation and wish to start over without reloading the knowledge base, you may use the restore command.

review_frame

The review_frame command allows you to review any frame which is present in the Prolog database. It will prompt you for the frame's name. It will then display the contents of one of the frame's slot on the screen. Additional slots will be displayed by depressing the space bar.

review_goals

The review_goals command allows the displaying of all the goals which are currently in the Prolog database.

review_rules

The review_rules command allows the displaying of any of the rules which are currently in the Prolog database. Both the rule's premise and conclusion will be displayed.

rules

The rules command displays ABC's syntax rules. This command is primarily for the newer users.

save_wm

The `save_wm` command allows you to save the working memory portion of the knowledge base to a file. This will allow your knowledge base to learn and will prevent you from answering the same questions repeatedly from one consultation to another.

syntax

The `syntax` command is used if you are getting Prolog system errors when loading your knowledge base. This command will aide in detecting where syntax errors exist in your knowledge base. The `syntax` command reads your knowledge base file one term at a time and displays it on the screen. When you reach an error, if the error was caused by incorrect syntax, then the syntax problem is located in the term following the last successfully read term.

trace

The `trace` command is used after a consultation to determine the steps ABC took to derive its solutions. The steps can be viewed from the latest to the earliest one at a time by pressing the space bar. If you wish to abort the trace operation, press the Return or Enter key.

How To Create A Knowledge Base

Basic Structures in ABC - There are four basic structures used in ABC to build a knowledge base: the fact, the frame, the rule, and the askable. Just as in any programming language where there may be several ways to use the language's structures to get the same result, you can develop functionally equivalent knowledge bases in ABC using different combinations of structures. To continue this analogy, using the wrong structures in a programming language which produces inefficient or difficult to read code would be similar to using the wrong knowledge structure in ABC. It is therefore important to know each of the structures in ABC and to know when to properly use them.

Facts - Facts in ABC are used to assert a small amount of knowledge about some object. Specifically, the fact provides a known value for an attribute of the object. If the object has several relevant attributes and/or values, the fact structure might be a poor choice. If the object is related hierarchically with other objects, then the fact structure should be abandoned for the frame structure.

Facts are easy to write. To write a fact which states that the Math 555 course is difficult with a certainty of 95 can be seen in the example below.

fact: math555 is difficult cf 95.

If you do not need certainty factors in your knowledge base, you can leave them off your facts. Facts without an explicit certainty factor have a implicit certainty factor of 100.

Frames - The frame structure is appropriate when you have hierarchical relationships among objects. It may also be appropriate when an object has several attributes or several values for any given attribute.

Frames in ABC represent some object. The slots of a frame represent the attribute of the object. A frame can be linked hierarchically to other frames by having a slot called "ako" with its slot-value as a list of other frames which are higher in its hierarchical tree. An example of a frame which has an hierarchical value can be seen in the example below.

```
frame : tweetie
      slot 'ako value canary'
      slot 'owner value joe sherry'
      slot 'born if_needed ask_dob'
      slot 'age if_needed find_age'.
```

Example of a Hierarchical Frame

The basic structure of a frame can be seen in the example above. The frame-name or the object of the frame is "tweetie". The first slot is where the "ako" (a kind of) link is provided to the frame called "canary". The second slot has the name "owner" and has a slot-attribute of "value". The slot "owner" has two values: "joe" and "sherry". The third slot, called "born", has a slot-attribute of "if_needed". Whenever the slot-attribute is "value" or "default", then the slot-value is a list of values. But when the slot-attribute is either "if_needed",

"if_added", or "if_removed", the slot-value is a demon procedure. So in the third slot, ABC would execute the procedure "ask_dob" in order to acquire tweetie's birthyear.

If you develop a knowledge base which includes frames with demon procedures, you must make sure that you also write demon procedures in accordance with part five of this user's manual.

Rules - Rules are a way of expressing known relationships or heuristics (rules of thumb). In ABC rules take on the general format shown in the example below.

```
rule_18 : if  'main_component is poultry' and
              'meal includes turkey'
            then 'best_color is red cf 80'.
```

Example of A Rule in ABC

This rule can be interpreted fairly easily. If the main component of the meal is poultry and the meal does include turkey, then the best color for the meal's wine is red with a relative certainty of 80.

The best time to use rules is when the nature of the knowledge fits into the "If A and B and C THEN Z" structure.

Each rule must have a unique rule number which is not maintained by ABC.

Askables - There are two types of askables: initial and ordinary. They have a lot of similarities and only one difference. Askables are used when you want to get information from the end-user. If you are developing a knowledge base which requires the person's weight, height, and sex to determine risk due to coronary disease and it is not feasible to build a frames, then askables would probably be appropriate. To create an askable you must make up a straight-forward unambiguous question and be able to enumerate the valid solutions. ABC also imposes a limit of nine valid solutions per askable. An example of an askable can be shown in the example below.

askable : 'sauce is Sauce_Type' derived from
 "What kind of sauce is it?" and 'spicy sweet cream tomato'.

Example of An Askable

The structure of the askable starts with the key word "askable" or "initial_askable" (the only structural difference between an initial and ordinary askable), and is followed by a semicolon and a quoted OAV triple (the value of the triple may be a variable). If the value of the object-attribute pair is a variable, the variable becomes instantiated to the end-user's choice. Notice that the question is flanked by a set of double quotes and the list of valid solutions (spicy, sweet, cream, and tomato) are flanked by a set of single quotes and that the question and valid

solutions are separated by the key word "and". The way this askable will be displayed to the end-user can be shown in the example below.

What kind of sauce is it?

1. spicy
2. sweet
3. cream
4. tomato

Enter Number (or w for why) > 3

Example of An Askable Prompt

The "why-trace" is invoked by pressing the letter "w" instead of choosing a number.

How To Write Demon Procedures

Demon procedures in frames, written in standard Prolog, allow for automatic frame maintenance. If a frame called "john doe medical record" has the slots "doctor", "current treatment", "date last treated", along with several typical medical record slots, then a demon procedure to notify the doctor in the "doctor" slot when the current treatment is altered may be appropriate. Maybe a demon procedure to update the "date last treated" slot whenever the current treatment changes might be useful. Once these demon procedures are in place, the action which they perform is hidden from the end-user and thus costly mistakes can be avoided.

There are three types of demons in ABC: "if_needed", "if_added", and "if_removed". Each of these three types of demons are further described below.

The "if_needed" demon is used when the developer either can not or does not want to provide a value for an attribute of some object. If the value is subject to change with time (i.e. age), then rather than assigning a numerical value to the facet which will need continuous updating, it might benefit the developer to use a demon. If the facet is 'age', then a procedure to acquire the date and calculate the age may be preferable to placing the actual age into a 'value' facet. An example of a "if_needed" demon is shown on the following page.

```

find_age(Animal, [Age]) :-
    frame_get(Animal, born, [Birthyr]),
    nl, nl,
    write('What year is this? (yyyy): '),
    read_year(Year),
    Age is Year - Birthyr.

```

Notice in the example "if_needed" demon that the demon predicate has two arguments with the second argument being a list. This must be the case for any "if_needed" demon created if it is to operate properly in ABC.

The "if_added" demon is used when the developer wishes an action to be performed automatically, from the perspective of the end-user, whenever a value is added to a particular frame. The construction of a "if_added" demon procedure is identical to that of the "if_needed" procedure and thus will not be repeated.

The "if_removed" demon is used when the developer wants a certain action to be performed when a slot-value is removed from a slot. Again, the construction of a "if_removed" procedure is no different than either the "if_needed" or "if_added" procedures.

Note that all three types of demon procedures have to be written in standard Prolog. The developer may make use of predicates defined by ABC in defining his demon procedures.

A Typical Session With The Wine Knowledge Base

Step One Make sure that you have a copy of the files called "WINE.KB" and "WINE.WM" on the current disk drive. These two files are the static knowledge base and working memory for the wine advisor.

Step Two Execute the Prolog interpreter and consult ABC. Notice that the screen will display ABC's syntax rules and then provide the ABC prompt. Type in the ABC command "load". Notice that ABC now prompts you for a filename. Type in the filename "wine.kb". This tells ABC that you wish to load in the wine advisory knowledge base. After a brief delay, ABC should prompt you to see if you would like to also load in working memory. Just press the return key; ABC will read in the default reply of yes. The next two prompts which ABC will generate, the loading of an auxiliary file and starting a consultation, need to be answered by typing in "n" for "no".

There is no auxiliary file to read; trying to read a file which does not exist will result in a Prolog level or operating system level error.

The reason for not starting a consultation is that the knowledge base "wine.kb" has no preset goals, thus execution of any knowledge base without a goal will result in ABC not being able to generated a solution.

Step Three Type in the ABC command "review_goals" to prove to yourself that there are no goals in the Prolog database.

Step Four Now add a goal to the database by typing the ABC command "add_goal". Notice that ABC provides you with instructions on how to enter the goal and also prompts you with a different type prompt. Type in the goal "recommended_wine is Wine" without the quotes and press return. Notice that you are returned to the ABC level prompt.

Step Five Type in the ABC command "review_goals" again to see the new goal displayed. Notice that the variable has been replaced by some internal Prolog variable pointer.

Step Six Type in the ABC command "review_frame" to review one of the wine advisor's frames. Note that ABC will prompt you for the name of a frame. In response to this prompt, type in the frame name "chardonnay", again without the quotes. Notice that ABC will display the first slot (color), its associated facet (value) and the values of the slot (white). Press the space bar twice to review the other two slots of the frame and notice that the ABC level prompt is ready for your next instruction.

Step Seven Type in the ABC command "commands" and press return. Notice that ABC will display all of the ABC commands.

Step Eight Type in the ABC command "go" and press return to start a consultation session. Notice that ABC prompts you with a question about the main component of a meal. Let us assume that the main component of the meal is meat, so type in the number "1" which corresponds to the valid answer "meat".

ABC should now prompt you for whether the meal has a sauce or not. Let us assume that it does and reply accordingly. Type in the number "1".

Now you should see a prompt asking you for the flavor of the meal. Let us assume that we have an average meal and type in the number "2".

The next prompt should be asking you to enter what type of sauce is being used on the meal. Type in the letter "w" for why and notice how ABC explains why a question is being asked. Type in "n" when prompted for addition trace. Notice that the question is asked again. Let us assume that we will be using a tomato sauce and enter the number "4" in at the prompt.

You should now be prompted with the question of whether your meal has veal in it. Reply negatively by typing in the number "2".

The next prompt will ask you what level of sweetness you prefer in your wine. Type in the number "3" corresponding to the answer "sweet". Notice that after a short delay, that the solutions are displayed on your monitor. You should have three different types of wines being suggested for your meal each having a certainty factor associated with it. The wine Gamay had the greatest certainty factor ($cf = 60$), and can thus be considered as the best wine for the meal.

Step Nine Press any key to continue. Notice the prompt about saving working memory. Type in "n" for "no" and you should be back at the ABC level prompt.

Step Ten Type in the ABC command "trace" and press the return key. Notice a goal being displayed. Press the space bar several times to continue displaying the trace and then press the return key to exit the trace before it completes. Reenter the ABC command "trace" and press the space bar repeatedly until the trace is complete. Notice that obtaining the trace can be accomplished several times to any point and then terminated without removing the trace from the Prolog database.

Step Eleven Now that the consultation is over lets remove all the information learned from this consultation in preparation for another consultation. Type in "restore" to restore the Prolog database and answer the following safety prompt affirmatively. Now type in the ABC prompt "trace" to see what is in the trace mechanism. Notice that the trace is empty.

Step Twelve Now you are ready to start a new consultation, by typing "go", or exit ABC by typing the command "quit".

If you cannot remember an ABC command, remember to type in the command "commands" or "help".

Appendix D: ABC Predicates

This appendix is a reference guide to all the ABC defined predicates. Each predicate in this appendix will have a brief description of their function along with any constraints which may be imposed for proper operation. While some safeguards have been made to prevent these predicates from not behaving erratically when they are used outside of their constraints, it was impossible to completely safeguard against user-induced errors because Prolog has very little error-exception capability.

The ABC predicates are listed in alphabetical order. Any of these predicates can be used in user-defined predicates or demons provided ABC is consulted prior to their use.

add_value /5

The add_value predicate adds a value to a slot when the frame is known to exist. If either the slot or the slot attribute is not present, then they will be created in order to hold the new value which is being added.

Constraints: add_value(+,+,+,+,-).

ask_initial_askables /0

The ask_initial_askables predicate searches the Prolog database for 'initial_askables' and if found, asks the user a question and prompts him for a reply. This will continue until all the 'initial_askables' are asked.

Constraints: All initial_askables in the Prolog database must be structured in accordance with the ABC user's manual.

ask_question /5

The ask_question predicate checks to see if the question from an askable can be asked. If the OAV triple associated with the question is already in the trace, then the question will not be allowed. The ask_question predicate sets up how the question is asked, gets the user's reply to the question, and asserts the appropriate OAV triple in the trace mechanism. It also allows the user to inquire as to "why" a question is being asked.

Constraints: ask_question(+,+,+,-,-).

assert goals /1

The `assert_goals` predicate asserts goals into the trace mechanism if the goals are solved.

Constraints: `assert_goals(+)`.

assert in trace /2

The `assert_in_trace` predicate asserts one of the ABC structures into the trace. If the structure already exists in the trace, then it will not be asserted a second time. If a fact, confirmed assertable, or a rule already exists with the same OAV triple as a solution, then the original OAV triple's CF will be adjusted to reflect the additional support but the new structure will not be placed into the trace. If neither of the above situations are present, then the structure is placed into the trace mechanism.

Constraints: `assert_in_trace(+)`. The argument must be a valid ABC structure (i.e. 'fact: [O,A,V]) or the rule number of a top level rule which solved the goal.

assert rule trace

The `assert_rule_trace` predicate stores a temporary trace in the Prolog database. If the rule succeeds in which this temporary trace is associated with, then the temporary trace is also asserted into the main 'why' trace. Each of these temporary traces are distinguished from other temporary traces by using the rule number and the condition number.

Constraints: `assert_rule_trace(+,+,+)`.

calculate CF /3

The calculate_CF predicate takes as its two input arguments, two certainty factors, which it uses to calculate the combining certainty factor based upon the formula $\text{Combined CF} = \text{CF1} + (100 - \text{CF1}) / 100 * \text{CF2}$.

Constraints: calculate_CF(+,+, -). The two inputs must be either lists, empty or with one number between and including zero to one-hundred, or just the number between with the same constraints. The two types cannot be mixed.

change /1

The change predicate controls the changing of the term read from the ".kb" format to the ".abc" format and writes the new format out to the new file with the new extension. For purposes of recursive programming, it then calls the makefile predicate which will read another term and the process is then repeated.

Constraints: change(+). Same constraints as the "makefile" predicate.

check /2

The check predicate checks a filename for the proper extension. If the extension of the filename provided is either ".kb" or ".abc" then the extension name is returned; otherwise, the "check" predicate fails.

Constraints: check(+, -).

check_ext /2

The `check_ext` predicate is used in conjunction with the "check" predicate. The `check_ext` predicate tries to match the extension of a filename with either "kb" or "abc" in the form of ASCII lists.

Constraint: `check_ext(+,?)`. The first argument must be a list.

circulate_trace

The `circulate_trace` predicate actually does all the work which the 'replace_trace' predicate is responsible for. It circulates the trace clauses in the Prolog database, 'popping off' clauses from the top of the stack and placing them back on the bottom, until the entire stack of trace structures have been moved back to their original position. Additionally, it replaces the CF of the old trace structure with the new CF in the shuffle.

Constraints: `circulate_trace(+)`. The argument must be a valid ABC knowledge structure.

clean_up_why_trace

The `clean_up_why_trace` predicate is necessary because when rules and their conditions are placed into the why trace, they are not removed if the rule fails. In such an instance, the why trace gets corrupted. This predicate increases the efficiency of the why trace by removing unnecessary assertions. It does this by requiring assertions to either be part of the goal itself, or part of the rule's condition which is just above it in the order of when it was asserted.

Constraints: `clean_up_why_trace(+,+,+)`. Arguments must be atoms.

cls /0

The cls predicate simply clears the current screen. In Arity Prolog, this predicate is built-in.

condition /2

The condition predicate takes triples which were read using ABC's readline predicate and converted from the quoted triple form to the list form and then writes them out to a temporary file called ABC.TMP. It then reads the file back into the Prolog database using the Prolog read predicate. This is necessary because words written beginning with an underscore or a capital letter is meant to be a variable but ABC's readline predicate cannot do this without using the Prolog reader.

Constraints: condition(+,-).

convert_filename /3

The convert_filename predicate takes a filename and a new extension and returns a filename with the same prefix yet with the new extension.

Constraints: convert_filename(+,+,-).

convert_term /2

The convert_term predicate converts a user-friendly term to one which ABC can understand internally. The convert_term predicate is used only on ABC data structures. It will eventually replace all the single quoted triples such as 'amy loves john' to a list structure such as [amy, loves, john] which is what the inference mechanism of ABC requires.

Constraints: `convert_term(+,-)`. This predicate may not work on structures other than those structures define in ABC.

convert triple /2

The `convert_triple` predicate will parse through a quoted atom and tokenize the atom into a list of words. See the comments about the predicates "`tokenize /3`" and "`get_rest_word /4`" for a more detailed explanation.

Constraints: `convert_triple(+,-)`. The first argument must be a list of ASCII numbers.

convert to list /2

The `convert_to_list` predicate converts an atom to a list of words which the atom originally had separated by spaces. In ABC, this predicate is used to convert a quoted triple into a triple in which the object, attribute, and value are in the form of a list.

Constraints: `convert_to_list(+,-)`.

delete value /5

The `delete_value` predicate will delete a value from a slot and return a new list of slots if give the slot-name, slot-facet and the value which needs to be deleted along with the original list of slots.

Constraints: `delete_value(+,+,+,+,-)`.

digit /1

The digit predicate tests to see if an ASCII number symbolizes one of the digits between and including zero through nine. The predicate succeeds if it does symbolize a digit and fails if it does not symbolize a digit.

Constraints: digit(+). Same as the predicate 'letter'.

display intro screen /0

The display_intro_screen predicate simply displays the introductory message on the display device.

Constraints: None.

explain why /1

The explain_why predicate explains why a question is being asked to the user if he requests an explanation. It uses the ABC predicate 'justify' to assist in this utility.

Constraints: explain_why(+).

fdelete /4

The fdelete predicate will delete a particular value from a frame if the frame-name, slot-name, slot-facet, and value are provided.

Constraints: fdelete(+,+,+,+).

fget /4

The fget predicate is the utility predicate which actually does the work of the 'frame_get' predicate.

Constraints: fget(+,+,+,-).

fput /4

The fput predicate is the utility predicate which does the work for the frame_put predicate. Given a frame-name, a slot-name, a slot-facet, and a value, the fput predicate will place the value into that particular frame/slot/attribute combination if it exists, or creates it if it does not exist.

Constraints: fput(+,+,+,+).

frame get /3

The frame_get predicate retrieves a list of values from a frame. It goes through a predetermined search to obtain its values.

Constraints: frame_get(+,+,+).

frame put /3

The frame_put predicate puts a value into a frame. It first sees if there is a 'if_added' demon associated with the frame and executes the demon procedure if it is. Otherwise, it places the value into the frame and slot specified under the slot-attribute of 'value'. If the slot or frame does not exist, they are created.

Constraints: frame_put(+,+,+).

frame remove /2

The `frame_remove` predicate removes all the values for a particular frame and slot. This includes all slot-attributes. This deletion effectively deletes the slot.

Constraints: `frame_remove(+,+)`.

get answer /2

The `get_answer` predicate is given as its first argument, the maximum number which can be returned as a valid response to an askable. If the user tries to respond with an invalid answer, the `get_answer` predicate will flag the 'invalid input' and prompt the user for a valid answer. This predicate returns either the number which corresponds to the answer selected by the user or the atom 'why' which corresponds to the user wanting an explanation to the reason for the question.

Constraints: `get_answer(+,-)`. The first argument must be an integer between 1 and 9.

get last goal /3

The `get_last_goal` predicate simply breaks the goals list into the last goal and all the remaining goals.

Constraints: `get_last_goal(+,-,-)`.

get list /4

The `get_list` predicate assumes that `Obj-Attr-Val` triple is in the trace somewhere with a certainty factor less than 100. The `get_list` predicate will return a list of all values along with their associated certainties for every different value found in the trace which matches the `Obj-Attr` pair.

Constraints: `get_list(+,+,+,-)`. The first three arguments must be atoms.

get reply /1

The `get_reply` predicate gets the next character placed into the current input device and determines whether its a carriage return, a 'y', a 'n', or a 'w', or anything else. If the character was either a carriage return or a 'y', the `get_reply` will return the atom 'yes'. If the character was a 'n', it will return the atom 'no'. If the character was a 'w', it will return the atom 'why'. If the character was anything else, it will prompt the user to reenter the character.

Constraints: `get_reply(-)`.

get rest quote /4

The `get_rest_quote` predicate gets as its first argument a list of ASCII numbers representing the remaining portion of an AOV triple which it is trying to tokenize. The leading quote has already been seen and now `get_rest_quote` will get the remaining portion of the quote, convert it from an ASCII list to an atom, and return it as its third argument.

Constraints: `get_rest_word(+,+,--)`. The first argument must be a list of ASCII numbers. The second argument must be a list.

get_rest_word /4

The `get_rest_word` predicate is fundamentally the same as the `get_rest_quote` predicate. The `get_rest_word` predicate will be given an ASCII list L1, and a second list, L2, which may have ASCII numbers also, and will append all the ASCII numbers up to the first '32', representing a space, of L1 to L2, convert the list over to test and output it as Word and output the remainder of the ASCII numbers as REM.

Constraints: `get_rest_word(+,+,--)`. Same as `get_rest_quote`.

get_rule /4

The `get_rule` predicate will retrieve a demon procedure (rule) from the frame structure either from the 'Frame' and 'Slot' specified or from a parent/grandparent of the 'Frame' specified.

Constraints: `get_rule(+,+,+,-)`.

identifier /1

The `identifier` predicate identifies valid characters which may be inside ABC's OAV triples.

Constraints: `identifier(+)`. The argument must be an atom.

is known /4

The `is_known` predicate is the primary predicate which controls the inference characteristics of ABC. The `is_known` predicate is normally called upon by a rule to try to infer more implicit knowledge from explicit facts and rules. It also returns the certainty factor which the triple is known.

Constraints: `is_known(+,+,?,-)`. The structure of the first argument must be in the form of a ABC triple of an ABC rule condition.

known /2

The `known` predicate is activated when a top-level rule or frame matches a goal. It attempts to satisfy this top-level rule or frame by matching it against other facts, assertions, rules, frames or askables in the knowledge base. If the top level rule or frame is 'known', then the goal is said to have a solution. The 'known' predicate uses the 'is_known' predicate to solve the lower levels of the search for a solution. If a triple is 'known', the `known` predicate will succeed and will return a calculated certainty factor giving the relative strength of how well the OAV triple is 'known'.

Constraints: `known(+,-)`. The first argument must be an AOV triple.

letter /1

The `letter` predicate tests to see if an ASCII number symbolizes some letter of the alphabet. If it does, the 'letter' predicate succeeds. If not, the predicate fails.

Constraints: letter(+). The argument must be an ASCII number between 1 and 255.

load /1

The load predicate asserts a term into the Prolog database and calls the ABC predicate "loadfile" until there is no more terms to load, at which time, the "end_of_file" marker should be reached and the load predicate will simply succeed not allowing backtracking.

Constraints: load(+). The ABC predicate "loadfile" must exist and its constraints met.

loadfile /0

The loadfile predicate reads a term from a file and loads it into the Prolog database with the use of the load predicate. The load predicate actually asserts the term into the Prolog database and recursively calls loadfile until the end_of_file is seen and then terminates the loading process. This predicate is the same as in BC3.

Constraints: The calling procedure must "see" a valid Prolog file.

load abc /1

The load_abc predicate checks to see if the filename provided is an atom and then reconsults the file.

Constraints: load_abc(+). A valid filename must be provided as the input.

load aux /1

The load_aux predicate is used to load user-defined Prolog predicates into the Prolog database. This includes demons, if they exist.

Constraints: load_aux(+). The constraints for the load_aux predicate are the same as the "load_wm" predicate. An additional constraint imposed in this predicate is that all OAV triples must be in the form of a Prolog list.

load kb /1

The load_kb predicate reads an unparsed knowledge base a term at a time, parsing the term, converting quoted triples to triples within a list, and writing the converted triple to a new file. This new file will have the same prefix but it will have the extension of ".abc". Finally, this predicate will read all of the new terms located in the new file and assert them into the Prolog database.

Constraints: load_kb(+).

The entire file name must be passed to load_kb, including the path. Since this path/file name must be an atom, the path/file name must be surrounded by single quotes for proper operation. Additionally, if the filename does not exist, the system will normally produce an error message which may be impossible to recover from without rebooting the system and losing your data.

load_wm /1

The `load_wm` predicate takes the ".abc" or ".kb" knowledge base filename, creates a filename with the same prefix yet with an extension of ".wm". It then searches the current drive for the file with that name. The file is then read in much like the ".kb" file, parsed and converted, written back to a file called "ABC.TMP". Finally, the "ABC.TMP" file is opened and read using the Prolog reader and the terms are asserted into the Prolog database. The temporary file was necessary to get Prolog to accept variables as variables instead of quoted atoms. Constraints: Basically, the same as the "load_kb" predicate. The system will normally produce a system error if the working memory file does not exist.

makefile /0

The `makefile` predicate will read in a term from a file, parse the term and convert it so that all quoted OAV triples are converted to OAV triples within a Prolog list and then write them back out to a second file. The term is read using the Prolog read predicate and the outputting to the new file is done via ABC's change predicate. The calling procedure is responsible for "seeing" and "telling" the appropriate file.

Constraints: The calling procedure must "see /1" a valid Prolog formatted file to read from it and must also "tell /1" a file to open an output file. Note that if this file already exists, the "tell" predicate will overwrite existing data in favor of the new data.

not in trace /2

The `not_in_trace` predicate checks both the main trace and the temporary rule traces to see if a fact or rule has already been asserted with the same object and attribute pair. If there are no assertions found, the predicate will return successful, otherwise, it will fail.

Constraints: `not_in_trace(+,+)`. Both arguments must be atoms.

parse name /3

The `parse_name` predicate takes an atom (which is usually a filename) and divides it into two parts. The filename is parted into two ASCII lists, one representing the filename's prefix and the other representing the filename's extension.

print goals /1

The `print_goals` predicate displays the goal's list. It is used within ABC to display the goals during a trace. This predicate may be used to display any number of OAV triples which are in list form separated by the ABC defined operator 'and'.

Constraints: `print_goals(+)`.

print trace /0

The `print_trace` predicate actually lists the trace on the screen. It extracts from the top of the Prolog database stack and pretty prints each trace assertion one at a time. Pressing the space bar enables the next trace to be displayed. This loop continues until no more triples are left or until the user presses the ENTER key.

Constraints: None.

readline /2

The readline predicate allows the user to respond to a prompt without the requirement that the user's response be a proper Prolog term. It will allow letters and numbers and a few symbols to be entered in via the keyboard. It will also allow the deletion of the last character using the backspace-rubout key. It will return whatever is typed in as an atom once the return key is pressed.

Constraint: readline(+,-).

read_facet /1

The read_facet predicate safeguards the user from entering in a facet_type which is not recognizable by ABC.

Constraints: read_facet(-).

replace_trace

The replace_trace predicate replaces a clause in the trace mechanism with an updated clause. Actually all that gets updated is the certainty factor. Whenever a second rule or fact derives a solution which has already been found and placed into the trace, its certainty factor will be used to increase the CF of the original assertion.

Constraints: replace_trace(+). The argument must be a valid ABC knowledge structure.

restore kb /0

The `restore_kb` predicate retracts all the temporary knowledge of previous consultations which were inserted into the Prolog database.

Constraints: None.

reverse goals /2

The `reverse_goals` predicate simply reverses the goals list so that the goals may be asserted into the trace mechanism in their proper order. Because of the operator precedence problems, the reversed goals must be created by the last goal found in the original goal list and the reverse of the remainder of the original goals.

Constraints: `reverse_goals(+,-)`.

rules /0

The `rules` predicate displays the syntactical rules which must be obeyed throughout any consultation with ABC in order for it to perform correctly. The ABC command "rules" calls this predicate so the user may view the syntax rules at any ABC prompt.

Constraints: None.

save wm /0

The `save_wm` predicate saves the working memory portion of the knowledge base to a separate file. The filename is provided by the user. The file can easily be read and altered by an ASCII editor. The `save_wm` predicate will save all the frames along with all the confirmed and denied triples in the working memory of the Prolog database.

Constraints: None.

search trace /2

The search_trace predicate searches both the main trace and the rule trace to determine if a triple is present in either of the traces. If it is, the predicate will succeed with some CF.

Constraints: search_trace(+,-).

sget

The sget predicate is a recursive predicate which obtains the values of the given frame/slot/slot-face combination.

Constraints: sget(+,+,+,-).

solve /1

The solve predicate is used to solve 'goals' in the form of "goals : goal_1 and goal_2 and ... goal_n." where each goal is an OAV triple. The solve predicate also initiates the 'why' trace and displays all solved triples which are not explicitly told by the user.

Constraints: solve(+). Goals must take the form outlined in the ABC user's manual.

start /0

The start predicate "initiates" the database and keeps subsequent operations in an infinite loop via the repeat/0 and execute/1 predicates. This loop prompts the user for an ABC command. Each of these commands will ultimately fail (except "quit") causing backtracking to the repeat

predicate where the cycle repeats with another prompt. There are a dozen ABC commands which, relative to the start predicate, work on the principle of "side-effects." The start predicate is called by the Prolog interpreter upon consulting the ABC shell. You may exit the ABC shell by typing "quit" at the ABC prompt.

Constraints: None.

tokenize /3

The tokenize predicate receives as an argument, a list of ASCII numbers representing the O-A-V triple which was in quotes. This list is then tokenized into a list of words. Tokenize does this through the aid of the `get_rest_word` predicate. Tokenize gets all the words from within the quoted atom 'A' which are separated by a space, appends them to the list called 'List', and forms the new list called 'Newlist'. The tokenize predicate is an alteration of a predicate by the same name taken from Claudia Marcus' book "Prolog Programming", pages 203-210.

Constraints: `tokenize(+,+, -)`. The first argument must be a list of ASCII numbers. The second argument must be a list.

write_conditions /1

The `write_conditions` predicate displays (pretty prints) the conditions of a rule during a trace. It will print out any condition which ABC is capable of understanding.

Constraints: `write_conditions(+)`.

write confirmed triples /0

The `write_confirmed_triples` predicate is used by the '`save_wm`' predicate to save all the confirmed triples to working memory.

Constraints: None.

write denied triples /0

The `write_denied_triples` predicate is used by the '`save_wm`' predicate to save all the denied triples to working memory.

Constraints: None.

write frames /0

The `write_frames` predicate retrieves all of the frames in the Prolog's database and writes them to the current output device.

Constraints: None.

write goal /4

The `write_goal` predicate displays all of the solutions on the screen in a 'pretty' print.

Constraints: `write_goal(+,+,+,+)`.

write slots /1

The `write_slots` predicate takes all the slots of a given frame and 'pretty prints' them to the current output device.

Constraints: `write_slots(+)`. The argument must be in the form '[Slot, SAttr, SVal, ...] slot [...]'.

write valid answers /4

The `write_valid_answers` predicate is responsible for numerating and displaying all the valid answers in a 'pretty' format. It also acquires the number of the valid answers and then passes this information on so a check can be made to make sure the user does not try to select a number that does not correspond to a valid choice.

Constraints: `write_valid_answer(+,+,--,-)`. The first argument must be an atom. The second argument must be a list. The third argument must be an integer.

Appendix E: ABC Predicate Dependencies

This appendix shows all of the predicates which ABC uses and the dependencies they have on other ABC predicates. This appendix was useful throughout the development of ABC and was especially useful in a couple of the marathon debugging sessions.

The predicates are listed as they appear in the draft copy of the source code. Each predicate is followed by all the ABC predicates which are used to define it. The forward slash with the number after it indicates the "arity" of the predicate, or the number of arguments the predicate has.

```
start /0
    display_intro_screen /0
    restore_kb /0
    readline /2
    execute /1

display_intro_screen /0
    rules /0

rules /0

execute /1
    cls /0
    readline /2
    check /2
    load_abc /1
    load_kb /1
    get_answer /1
    load_wm /1
    load_aux /1
    ask_initial_askables /0
    solve /1
    reverse_goals /1
    assert_goals /1
    get_reply /1
    save_wm /0
    execute /1
    restore_kb_wc /0
```

```

    print_trace /0
    remove_kb /0
    print_goals /1
    write_conditions /1
    write_frame /1
    convert_triple /2
    condition /2
    get_goals /1
    read_facet /1
    remove_goal /3
    commands /0

load_kb /1
    check /2
    convert_filename /3
    makefile /0
    loadfile /0

load_abc /1
    loadfile /0

load_vm /1
    convert_filename /3
    makefile /0
    retract_all /1
    loadfile /0

load_aux /1
    convert_filename /3
    loadfile /0

convert_filename /3
    parse_name /3

parse_name /3

makefile /0
    change /1

change /1
    convert_term /2
    makefile /0

loadfile /0
    load /1

load /1
    loadfile /0

load_abc /1

```



```

check /2
    check_ext /2

check_ext /2
    kb_error_msg /0
    check_ext /2

kb_error_msg /0

convert_term /2
    convert_term /2
    convert_triple /2

convert_triple /2
    tokenize /3

tokenize /3
    identifier /1
    get_rest_word /4
    tokenize /3
    get_rest_quote /4

get_rest_quote /4
    get_rest_quote /4

get_rest_word /4
    identifier /1
    get_rest_word /4

identifier /1
    letter /1
    digit /1

letter /1

digit /1

ask_initial_askables /0
    ask_question /5
    assert_in_trace /1

ask_question /5
    not_in_trace /2
    write_valid_answers /4
    get_answer /2

write_valid_answers /4

get_answer /2

```

```

solve /1
    retract_all /1
    known /2
    get_list /4
    write_goal /4
    solve /1

known /2
    retract_all /1
    is_known /4
    assert_in_trace /1

assert_in_trace /2
    trace /1
    calculate_CF /3
    replace_trace /2
    assert_in_trace /2

calculate_CF /3

get_list /4
    retract_all /1
    temp_list /1
    member /3

write_goal /4

reverse_goals /2
    get_last_goal /3
    reverse_goals /2

get_last_goal /3
    get_last_goal /3

assert_goals /1
    assert_trace /1
    assert_goals /1

save_wm /0
    cls /0
    readline /2
    write_frames /0
    write_confirmed_triples /0
    write_denied_triples /0

write_frames /0
    write_slots /1

write_slots /1
    writelist /1
    write_slots /1

```

```

write_confirmed_triples /0

write_denied_triples /0

restore_kb_wc /0
    get_reply /1
    restore_kb /0

restore_kb /0
    retract_all /1

print_trace /0
    writelist /1
    write_conditions /1

remove_kb /0
    restore_kb /0
    retract_all /1

write_frame /1
    write_frame /1

condition /2

get_goals /1

read_facet /1
    readline /2
    read_facet /1

remove_goal /3
    remove_goal /3

commands /0
    cls /0

is_known /4
    is_known /4
    retract_temp_rules /1
    assert_rule_trace /3
    frame_get /3
    retract_all /1
    not_in_trace /2
    ask_about /3
    assert_trace /1

retract_temp_rules /1
    retract_temp_rules /1

search_trace /2

not_in_trace /2

```

```
explain_why /1
  justify /2
  get_reply /1
  explain_why /1

justify /2
  writelist /1

frame_get /3
  fget /4
  frame_get /3

frame_put /3
  get_rule /4
  fput /4

frame_remove /2
  fdelete /4

fget /4
  sget /4
  member /2

fput /4
  add_value /5

add_value /5
  add_value /5

fdelete /4
  delete_value /5

delete_value /5
  remove /3
  delete_value /5

get_rule /4
  fget /4
  get_rule /4

print_goals /1
  print_goals /1

write_conditions /1
  writelist /1
  write_conditions /1

convert_to_list /2
  tokenize /3
```

get_reply /1
readline /2
 reverse /2
 readline /2

reverse /2
 append /3
 reverse /2

remove /3
 remove /3

writelist /1
 writelist /1

member /2
 member /2

retract_all /1

append /3
 append /3

cls /0

```

/*****
/*
/*                                PETS.KB                                */
/*                                05 Aug 88 / 26 Sep 88                    */
/*
/* This knowledge base was developed to demonstrate the functional- */
/* ity of several aspects of the AFIT Backward Chainer (ABC) expert */
/* system shell. In addition, it may be used as an example of how */
/* to write frames and rules for use with the ABC shell. There are */
/* two parts of most knowledge bases: the rulebase and the working */
/* memory. The rulebase contains all the rules, facts, initial */
/* askables, and goals (if any). The rulebase is the static portion */
/* of the knowledge base. The working memory contains the confirmed */
/* and denied triples along with the frames. The working memory is */
/* the dynamic part of the knowledge base and thus may grow through */
/* its use in consultations simulating "learning". This learning is */
/* can be accomplished without the end user even being aware of it. */
/*
/* The rulebase file has the extension of ".kb" or ".abc" depending */
/* on whether its been parsed or not. The working memory is kept in */
/* a seperate file with the same prefix yet with the extension of */
/* ".vm". When saving a rulebase, you will normally want to save it */
/* using this convention; otherwise, ABC will not see your working */
/* memory file and you may receive an error warning.
/*
/*-----

```

```

/*----- Facts and Rules -----*/

```

```

fact : 'joe loves tweetie'.
fact : 'amy loves rover'.
fact : 'joe isa male'.
fact : 'amy isa female'.

```

```

initial_askable : 'joe likes amy ' derived_from
"Does Joe like Amy? " and 'yes no'.

```

```

rule_1 : if    'Animal age Age'      and
               'Age >= 4'            and
               'Animal owner Owner'  and
               'Owner loves Animal'
            then 'Owner get_checkup Animal'.

```

```

rule_2 : if    'Animal_1 owner Owner_1' and
               'Owner_1 isa male'       and
               'Animal_2 owner Owner_2' and
               'Owner_2 isa female'     and
               'Owner_1 likes Owner_2'
            then 'Animal_1 will_see Animal_2'.

```

```

goals : 'joe get_checkup tweetie' and
        'tweetie will_see rover' and

```

'rover coat Coat'.

```
/* You must initially load this knowledge base using the ABC command */
/* "load." This will generate a new file on your disk called          */
/* PETS.ABC which is the file actually loaded in working memory.      */
/* A second file, ABC.TMP, will also be made but may be deleted later.*/

/* Run a consultation of this knowledge base with the above goals.    */
/* Afterwards, perform a trace and see what the trace mechanism in     */
/* ABC provides. Additionally, once you exit ABC, look at the file      */
/* PETS.ABC and see what your Prolog interpreter is actually reading    */
/* into working memory.                                                 */
```

```

/*****
/*
/*                                PETS.WM                                */
/*
/* This is the working memory part of the knowledge base called      */
/* pets. It should be entered using a text processor under a file    */
/* called "pets.wm".                                                  */

/*----- A Frame named Tweetie -----*/

frame : tweetie
  slot 'ako value canary'      /* tweetie is a kind_of canary. */
  slot 'owner value joe'      /* joe is tweetie's owner.    */
  slot 'born if_needed ask_dob' /* Execute demon ask_dob if you */
                               /* need tweetie's year of birth. */
  slot 'age if_needed find_age'. /* Execute demon find_age if you */
                               /* need tweetie's age. Notice */
                               /* also the period at the end */
                               /* signifying the end of frame. */

/*----- A Frame named Rover -----*/

frame : rover
  slot 'ako value dog'      /* Notice that what's inside the */
  slot 'owner value amy'    /* single quotes is the slot-    */
  slot 'born value 1979'    /* name followed by a space and */
  slot 'born if_needed ask_dob' /* then the slot-facet followed */
  slot 'age if_needed find_age'. /* by another space and finally */
                               /* the slot-value.              */

/*----- A Frame named Canary -----*/

frame : canary
  slot 'ako value bird'      /* Hierarchy is accomplished via */
  slot 'eats default seed'   /* the slot called ako which     */
  slot 'color default yellow'. /* always has a facet-value of   */
                               /* value or default.            */

/*----- A Frame named Dog -----*/

frame : dog
  slot 'ako value mammal'    /* The facet-values are limited */
  slot 'color default brown'. /* to: value, default, if_needed, */
                               /* if_added, and if_removed.    */

```


/*----- A Frame named Mammal -----*/

```
frame : mammal          /* The facet-values that start */
  slot 'ako value animal' /* with "if" have rules associ- */
  slot 'coat default hair'. /* ated with them which are exe- */
                          /* cuted when the slot is called.*/
```

/*----- A Frame named Animal -----*/

```
frame : animal
  slot 'num_legs default 4'
  slot 'num_eyes value 2'.
```

/*----- Demons for the above frames -----*/

```
/*
/* Demons must be written in Prolog. You may make use of any pred- */
/* icates already built-in and defined by Clocksin and Mellish or */
/* any predicates already defined by the ABC shell. The frame base */
/* language used by ABC has several very important conventions when */
/* writing demons. Please consult the ABC user's manual for details. */
/*
/* Demons along with all other self-defined Prolog predicates must */
/* be defined in an auxiliary file. Using ABC convention, the name */
/* of this auxiliary file will have the same prefix as your ".kb" */
/* file, yet with the extension ".aux". This convention will make */
/* it easier to remember. One trick to keep from remembering the */
/* auxiliary filename is to "reconsult" it from within your working */
/* memory or rulebase files. */
```

```

/*****
/*
/*                      PETS.AUX                      */
/*
/* This is the auxiliary file which contains any Prolog code which */
/* includes any demons written for the PETS knowledge base. The two */
/* demons, "find_age" and "ask_dob" are both defined in this file. */
/*
/*-----*/

```

```

find_age(Animal,[Age]) :-          /* Get the animal's birthyear */
    frame_get(Animal,born,[Birthyr]), /* from the frame structure, get */
    nl,nl,
    write('What year is this? (yyyy): '), /* the current year, and */
    read_year(Year),                /* subtract and find age. */
    Age is Year - Birthyr.

```

```

/* Notice that the frame base language predicate called "frame_get/3" */
/* was used in the above demon. The ABC predicate "is_known/4" */
/* could have been used more generally, searching the trace, facts, */
/* rules, etc before searching the frames. But the assumption was */
/* made that associated knowledge will be grouped together, one of */
/* the fundamental purposes of frames, so "frame_get/3" was the more */
/* efficient and practical choice.

```

```

ask_dob(Animal,[Birthyear]) :-    /* After the birthyear is found, */
    nl,nl,                        /* the frame base language pred- */
    write('What year was '),      /* icate "frame_put/3" is used */
    write(Animal),                /* to assert this knowledge into */
    write(' born? (yyyy): '),     /* the frame structure so that */
    read_year(Birthyear),         /* the user will not be bothered */
    frame_put(Animal,born,Birthyear). /* with this query again. Also, */
                                    /* this knowledge can be saved */
                                    /* with the rest of working */
                                    /* memory.

```

```

read_year(Year) :-
    readline([],Reply),
    nl,
    ( integer(Reply),
      Year = Reply,
      Year > 1950, !
    ;
      write('Please re-enter year, e.g., 1980: '),
      read_year(Year)
    ).

```

```

/*****
/*
/*                               WINE.ARI                               */
/*                               25 Aug 88/29 Aug 88                       */
/*                               */
/* This is a knowledge base which is designed to run on the ABC ex- */
/* pert system shell. It was made specifically to show the */
/* functionality and usefulness of the ABC shell. The wine know- */
/* ledge base is similar to Teknowledge's WINE knowledge system. */
/*                               */
/* ----- */

```

initial_askable : 'main_component is Main Component' derived from
 "What is the main component of the meal?" and 'meat fish poultry'.

initial_askable : 'meal has sauce' derived from
 "Does the meal have a sauce on it?" and 'yes no'.

askable : 'sauce is Sauce_Type' derived from
 "What kind of sauce is it?" and 'spicy sweet cream tomato'.

initial_askable : 'tastiness is Tastiness' derived from
 "What is the flavor of the meal?" and 'delicate average strong'.

askable : 'best_body is Preferred_Body' derived from
 "What type of body do you prefer your wine to have?" and
 'light medium full'.

askable : 'best_color is Preferred_Color' derived from
 "What color of wine do you prefer?" and 'red white'.

askable : 'best_sweetness is Preferred_Sweetness' derived from
 "What level of sweetness do you prefer in a wine?" and
 'dry medium sweet'.

rule_1 : if 'meal has sauce' and
 'sauce is spicy'
 then 'best_body is full'.

rule_2 : if 'tastiness is delicate'
 then 'best_body is light cf 80'.

rule_3 : if 'tastiness is average'
 then 'best_body is light cf 30'.

rule_4 : if 'tastiness is average'
 then 'best_body is medium cf 60'.

rule_5 : if 'tastiness is average'
 then 'best_body is full cf 30'.

rule_6 : if 'tastiness is strong'

then 'best_body is medium cf 40'.
 rule_7 : if 'tastiness is strong'
 then 'best_body is full cf 80'.
 rule_8 : if 'meal has sauce' and
 'sauce is cream'
 then 'best_body is medium cf 40'.
 rule_9 : if 'meal has sauce' and
 'sauce is cream'
 then 'best_body is full cf 60'.
 rule_10 : if not 'main component is poultry'
 then 'meal includes turkey cf 0'.
 askable : 'meal includes turkey' derived from
 "Does the meal have turkey in it?" and 'yes no'.
 rule_11 : if not 'main component is meat'
 then 'meal includes veal cf 0'.
 askable : 'meal includes veal' derived from
 "Does the meal have veal in it?" and 'yes no'.
 rule_12 : if 'meal has sauce' and
 'sauce is spicy'
 then 'feature is spiciness'.
 rule_13 : if 'main component is meat' and
 not 'meal includes veal'
 then 'best_color is red cf 90'.
 rule_14 : if 'main component is poultry' and
 not 'meal includes turkey'
 then 'best_color is white cf 90'.
 rule_15 : if 'main component is poultry' and
 not 'meal includes turkey'
 then 'best_color is red cf 30'.
 rule_16 : if 'main component is fish'
 then 'best_color is white'.
 rule_17 : if not 'main component is fish' and
 'sauce is tomato'
 then 'best_color is red'.
 rule_18 : if 'main component is poultry' and
 'meal includes turkey'
 then 'best_color is red cf 80'.

rule_19 : if 'main_component is poultry' and
 'meal includes turkey'
 then 'best_color is white cf 50'.

rule_20 : if 'meal has sauce' and
 'sauce is cream'
 then 'best_color is white cf 40'.

rule_21 : if 'meal has sauce' and
 'sauce is sweet'
 then 'best_sweetness is sweet cf 90'.

rule_22 : if 'meal has sauce' and
 'sauce is sweet'
 then 'best_sweetness is medium cf 40'.

rule_23 : if 'best_body is Best_Body' and
 'best_color is Best_Color' and
 'best_sweetness is Best_Sweet', and
 'Wine body Best_Body' and
 'Wine color Best_Color' and
 'Wine sweetness Best_Sweet'
 then 'recommended_wine is Wine'.

```

/*****
/*
/*           Working Memory for Wine.kb           */
/*           26 Aug 88                             */
/*
/* This data is listed in Teknowledge's M.1 Sample Knowledge Systems */
/* manual on page 3-22 and 3-23 in the form of M.1 rules.  The data */
/* has been represented here in frames to further demonstrate how to */
/* build frames to run with the ABC shell.                        */
/*                                                                 */
/* ----- */

```

```

frame : gamay
  slot 'color value red'
  slot 'body value medium'
  slot 'sweetness value medium sweet'.

```

```

frame : chablis
  slot 'color value white'
  slot 'body value light'
  slot 'sweetness value dry'.

```

```

frame : sauvignon blanc
  slot 'color value white'
  slot 'body value medium'
  slot 'sweetness value dry'.

```

```

frame : chardonnay
  slot 'color value white'
  slot 'body value medium full'
  slot 'sweetness value dry medium'.

```

```

frame : soave
  slot 'color value white'
  slot 'body value light'
  slot 'sweetness value dry medium'.

```

```

frame : riesling
  slot 'color value white'
  slot 'body value light medium'
  slot 'sweetness value medium sweet'.

```

```

frame : chenin blanc
  slot 'color value white'
  slot 'body value light'
  slot 'sweetness value medium sweet'.

```

```

frame : valpolicella
  slot 'color value red'
  slot 'body value light'
  slot 'sweetness value dry medium sweet'.

```

frame : cabernet sauvignon
slot 'color value red'
slot 'body value light medium full'
slot 'sweetness value dry medium'.

frame : zinfandel
slot 'color value red'
slot 'body value light medium full'
slot 'sweetness value dry medium'.

frame : pinot_noir
slot 'color value red'
slot 'body value medium'
slot 'sweetness value medium'.

frame : burgundy
slot 'color value red'
slot 'body value full'
slot 'sweetness value dry medium sweet'.

APPENDIX H

Expert System Examples Using M.1

This appendix contains two examples of expert systems developed by Teknowledge to demonstrate their M.1 expert system shell. Both of the expert systems are wine advisors but they are both presented to show the difference between how the knowledge looks. The first expert system called VINE uses the default operators while the second expert system CWINE uses the optional, user generated, operators to help make the knowledge base more readable.

<M.1> V I N E
Version 1.1
Copyright (c) Teknowledge Inc, 1984
August 16, 1984

This knowledge base contains the same knowledge as WINE, but uses logical variables (denoted by capital letters) so that syntactically identical rules from the WINE knowledge base can be collapsed into a single rule in VINE.

In addition, VINE uses logical variables in conjunction with a small database of wines, and a single 'table lookup' rule to find the wines, rather than a separate rule for each wine.

The top-level goal of the consultation is 'wine'.

goal = wine.

/* Before the system attempts any inferences,
the user's preferences are obtained: */

initialdata = [preferred-color, preferred-body, preferred-sweetness].

/* ----- BEST-BODY ----- */

/* The following rules use information about the sauce on the meal
and the meal's tastiness to find the best body for the wine. */

- rule-1: if has-sauce and
sauce = spicy
then best-body = full.
- rule-2: if tastiness = delicate
then best-body = light cf 80.
- rule-3: if tastiness = average
then best-body = light cf 30 and
best-body = medium cf 60 and
best-body = full cf 30.
- rule-4: if tastiness = strong
then best-body = medium cf 40 and
best-body = full cf 80.
- rule-5: if has-sauce and
sauce = cream
then best-body = medium cf 40 and
best-body = full cf 60.

/* ----- BEST-COLOR ----- */

/* The following rules use information about the main component
of the meal and the sauce on the meal to determine the best
color of wine to accompany the meal. */

- rule-6: if main-component = meat and
has-veal = no
then best-color = red cf 90.
- rule-7: if main-component = poultry and
has-turkey = no
then best-color = white cf 90 and
best-color = red cf 30.
- rule-8: if main-component = fish
then best-color = white.
- rule-9: if not(main-component = fish) and
has-sauce and
sauce = tomato
then best-color = red.

rule-10: if main-component = poultry and
has-turkey
then best-color = red cf 80 and
best-color = white cf 50.

rule-11: if main-component is unknown and
has-sauce and
sauce = cream
then best-color = white cf 40.

/* ----- BEST-SWEETNESS ----- */

/* The only rule that can help provide information about how sweet
the recommended wines should be 'fires' when the meal has a sweet
sauce on it. */

rule-12: if has-sauce and
sauce = sweet
then best-sweetness = sweet cf 90 and
best-sweetness = medium cf 40.

/* ----- DEFAULT-X ----- */

/* These default expressions are used when M.1 is unable to find
values for either best-CHARACTERISTIC or preferred-CHARACTERISTIC.

The single 'noautomaticquestion' entry keeps M.1 from ever asking
the user to provide a value for a default characteristic. */

noautomaticquestion(default-X).

default-body = medium.

default-color = red cf 50.
default-color = white cf 50.

default-sweetness = medium.

```

* ----- FEATURE ----- */

/* 'Feature' is a special characteristic of wine, currently used
   only to indicate that the meal is spicy. */

multivalued(feature).

    rule-13: if has-sauce and
    sauce = spicy
    then feature = spiciness.

/* ----- HAS-SAUCE ----- */

question(has-sauce) = 'Does the meal have a sauce on it?'.
legalvals(has-sauce) = [yes, no].

/* ----- HAS-TURKEY ----- */

question(has-turkey) = 'Does the meal have turkey in it?'.
legalvals(has-turkey) = [yes, no].

/* ----- HAS-VEAL ----- */

question(has-veal) = 'Does the meal have veal in it?'.
legalvals(has-veal) = [yes, no].

/* ----- MAIN-COMPONENT ----- */

multivalued(main-component).

question(main-component) =
    'Is the main component of the meal meat, fish or poultry?'.
legalvals(main-component) = [meat, fish, poultry].

/* ----- PREFERRED-BODY ----- */

multivalued(preferred-body).

question(preferred-body) =
    'Do you generally prefer light, medium or full bodied wines?'.
legalvals(preferred-body) = [light, medium, full].

```

/* ----- PREFERRED-COLOR ----- */

multivalued(preferred-color).

question(preferred-color) =
 'Do you generally prefer red or white wines?'.

legalvals(preferred-color) = [red, white].

/* ----- PREFERRED-SWEETNESS ----- */

multivalued(preferred-sweetness).

question(preferred-sweetness) = 'Do you generally prefer dry, medium or
sweet wines?'.

legalvals(preferred-sweetness) = [dry, medium, sweet].

/* ----- RECOMMENDED-X ----- */

/* These rules contain logical variables (in capital letters)
that make these rules applicable when seeking best-color,
best-body, and best-sweetness.

If best-CHARACTERISTIC is known, then that is what's recommended.
If the users preference about a particular characteristic is known,
then that's used as the recommended characteristic. If neither
the best nor the preferred characteristic is known, then a default
value is used. */

v-rule-1: if best-X = V
 then recommended-X = V.

v-rule-2: if best-X is unknown and
 preferred-X = V
 then recommended-X = V.

v-rule-3: if best-X is unknown and
 preferred-X is unknown and
 default-X = V
 then recommended-X = V.

```

/* ----- SAUCE ----- */

multivalued(sauce).

question(sauce) =
    'Is the sauce for the meal spicy, sweet, cream or tomato?'.

legalvals(sauce) = [spicy, sweet, cream, tomato].

/* ----- TASTINESS ----- */

multivalued(tastiness).

question(tastiness) =
    'Is the flavor of the meal delicate, average or strong?'.

legalvals(tastiness) = [delicate, average, strong].

/* ----- WINE ----- */

multivalued(wine).
multivalued(wine(COLOR,BODY,SWEETNESS)).

/* The following rule, with the table entries that follow, replaces the
bulk of the rules from the wine KB that conclude wine: */

v-rule-4: if recommended-color = C and
           recommended-body = B and
           recommended-sweetness = S and
           wine(C,B,S) = W
           then wine = W.

/* This 'noautomaticquestions' statement ensures that no automatic
question will be generated for wines not included in this table. */

noautomaticquestion(wine(COLOR,BODY,SWEETNESS)).

```

/* These table entries represent the mapping of attribute triples to specific wines. Note the use of the variable ANY in some entries to indicate that a particular attribute doesn't play a role in the selection of that wine: */

```
wine(red,medium,medium) = gamay.  
wine(red,medium,sweet) = gamay.  
wine(white,light,dry) = chablis.  
wine(white,medium,dry) = 'sauvignon blanc'. wine(white,medium,dry) =  
chardonnay.  
wine(white,medium,medium) = chardonnay.  
wine(white,full,dry) = chardonnay.  
wine(white,full,medium) = chardonnay.  
wine(white,light,dry) = soave.  
wine(white,light,medium) = soave.  
wine(white,light,medium) = riesling.  
wine(white,light,sweet) = riesling.  
wine(white,medium,medium) = riesling.  
wine(white,medium,sweet) = riesling.  
wine(white,light,medium) = 'chenin blanc'.  
wine(white,light,sweet) = 'chenin blanc'.  
wine(red,light,ANY) = valpolicella.  
wine(red,ANY,dry) = 'cabernet sauvignon'.  
wine(red,ANY,dry) = zinfandel.  
wine(red,ANY,medium) = 'cabernet sauvignon'.  
wine(red,ANY,medium) = zinfandel.  
wine(red,medium,medium) = 'pinot noir'.  
wine(red,full,ANY) = burgundy.
```

/* The following rule mentions feature, and hence is inconvenient to replace by a table-entry. */

```
rule-14: if recommended-color = white and  
         recommended-body = full and  
         feature = spiciness  
         then wine = gewuerztraminer.
```

C W I N E

Version 1.1

This knowledge base is in preliminary form,
it should not be considered 'expert'.

Copyright (c) Teknowledge Inc, 1984

August 17, 1984

- /* This knowledge base contains the same knowledge as WINE, yet
uses a cyclic control structure that allows multiple recommendations
to be made.

After a set of wines is recommended, the system asks the user if
those wines are satisfactory. If they are, then the consultation is
over. If the user indicates that the wines are NOT satisfactory,
the system asks which characteristics of the wine (color, body, or
sweetness) the user would like to change. After asking for the value
of the new characteristic(s), the system makes new recommendations.
This process continues until the user indicates that the wines are
indeed satisfactory.

This knowledge base introduces three new M.1 concepts:

- * user-modified syntax
- * presuppositions
- * mostlikely

User-modified syntax is a mechanism by which the knowledge
engineer can improve the readability of the knowledge base.
The KE can declare particular words as prefix, infix, or postfix
'operators'. These allow non-atomic expression names to be specified
without hyphens. For example, 'the-best-sweetness' can be written as
'the best sweetness' if 'the' and 'best' are declared as prefix
operators.

Presuppositions can be used by the knowledge engineer to specify that
a particular expression must be 'true' in order for it to make sense
to even seek another expression. For example, it wouldn't make sense
for the system to ask what kind of sauce is on the meal unless there
actually IS a sauce on the meal. This relationship between 'has
sauce' and 'sauce' can be explicitly represented by a presupposition
of the form

presupposition(sauce) = has sauce.

and does away with the need for various 'screening clauses' in rules
that test for sauce, which would otherwise be necessary.

The 'mostlikely' function can be used in the premises of rules to
find the value of a particular expression that is believed with the
highest certainty.

For example, 'mostlikely(wine) = X' will bind X to 'zinfandel'

```

    if that is the value of 'wine' that has the highest cf. */

/* The following words are used as operators in CWINE: */

prefix best.
prefix cycle.
prefix default.
prefix has.
prefix main.
prefix new.
prefix preferred.
prefix recommended.
prefix selection.
prefix the.
prefix user.

infix for.
infix with.
infix to.
infix of.

/* There is no 'goal' specification in the CWINE knowledge base.
   The top level goal for the consultation ('the consultation
   is over') is specified as an 'initialdata' expression so that
   M.1 will not automatically display its value at the end
   of the consultation. */

initialdata = [the consultation is over].

/* ----- CYCLE N IS COMPLETE -----*/

/* These rules are used to determine whether or not a particular
   cycle of recommendations is complete.

   A cycle is considered complete if either:

       * the wines have been determined,
         the wines have been displayed to the user, and
         the user is happy with those wines,
   or
       * no wines are found to be appropriate for the user */

rule-1: if the wine for cycle N is known and
        user informed of selection N and
        user happy with selection N
        then cycle N is complete.

```



```
rule-2: if the wine for cycle N is unknown and
        display(['Sorry, I'm unable to recommend any appropriate wines.',
                nl])
        then cycle N is complete.
```

```
/* If a particular cycle is complete, then the previous cycle is
   also complete. */
```

```
rule-3: if nextcycle to M = N and
        cycle N is complete
        then cycle M is complete.
```

```
/* ----- FEATURE ----- */
```

```
/* 'Feature' is a special characteristic of the wine, that's
   currently only used to indicate a particularly spicy meal.
```

```
   This attribute can have more than one value with certainty
   at a time. */
```

```
multivalued(feature).
```

```
rule-4: if sauce = spicy
        then feature = spiciness.
```

```
/* ----- HAS SAUCE ----- */
```

```
question(has sauce) =
    'Does the meal have a sauce on it?'.
legalvals(has sauce) =
    [yes, no].
```

```
/* ----- HAS TURKEY ----- */
```

```
/* If the main component of the meal is not poultry, then it's
   reasonable to assume that it doesn't contain turkey. */
```

```
rule-5: if not(the main component = poultry)
        then has turkey = no.
```

```
question(has turkey) =
    'Does the meal have turkey in it?'.
legalvals(has turkey) =
    [yes, no].
```

```

/* ----- HAS VEAL ----- */

/* Likewise, if the main component of the meal isn't meat,
   then it's reasonable to assume that it doesn't contain veal. */

rule-6: if not(the main component = meat)
        then has veal = no.

question(has veal) =
    'Does the meal have veal in it?'.
legalvals(has veal) =
    [yes, no].

/* ----- NEXTCYCLE TO M ----- */

/* This rule establishes the 'number' of the cycle that comes after
   cycle M. This is used when CWINE finds it necessary to create another
   cycle of recommendations. */

rule-7: if M + 1 = N
        then nextcycle to M = N.

/* ----- PREVIOUSCYCLE TO M ----- */

/* This rule finds the number of the cycle that immediately precedes
   the current cycle, so that M.1 can carry over the characteristics
   of wine from the previous recommendation cycle that the user didn't
   find objectionable. */

rule-8: if M > 1 and
        M - 1 = N
        then previouscycle to M = N.

/* ----- SAUCE ----- */

presupposition(sauce) = has sauce.

multivalued(sauce).

question(sauce) =
    'Is the sauce for the meal spicy, sweet, cream or tomato?'.

legalvals(sauce) =
    [spicy, sweet, cream, tomato].

```

```

/* ----- TASTINESS ----- */

multivalued(tastiness).

question(tastiness) =
    'Is the flavor of the meal delicate, average or strong?'.

legalvals(tastiness) =
    [delicate, average, strong].

/* ----- THE BEST BODY ----- */

/* These rules are used to find the best body of the wines to recommend,
   using the meal's tastiness and sauce (if it has one). */

rule-9: if sauce = spicy
        then the best body = full.

rule-10: if tastiness = delicate
         then the best body = light cf 80.

rule-11: if tastiness = average
         then the best body = light cf 30 and
           the best body = medium cf 60 and
           the best body = full cf 30.

rule-12: if tastiness = strong
         then the best body = medium cf 40 and
           the best body = full cf 80.

rule-13: if sauce = cream
         then the best body = medium cf 40 and
           the best body = full cf 60.

/* ----- THE BEST COLOR ----- */

/* These rules help find the best color of wine to recommend, based
   on characteristics of the meal itself. */

rule-14: if the main component = meat and
         has veal = no
         then the best color = red cf 90.

rule-15: if the main component = poultry and
         has turkey = no
         then the best color = white cf 90 and
           the best color = red cf 30.

rule-16: if the main component = fish
         then the best color = white.

```

```

rule-17: if not(the main component = fish) and
        sauce = tomato
        then the best color = red.

rule-18: if the main component = poultry and
        has turkey
        then the best color = red cf 80 and
        the best color = white cf 50.

rule-19: if the main component is unknown and
        sauce = cream
        then the best color = white cf 40.

/* ----- THE BEST SWEETNESS ----- */

/* If the meal has a sweet sauce, then a sweet-to-medium-sweet wine
   is probably appropriate. */

rule-20: if sauce = sweet
        then the best sweetness = sweet cf 90 and
        the best sweetness = medium cf 40.

/* ----- THE CONSULTATION IS OVER ----- */

/* This is only rule that concludes about the top level goal
   of the consultation. */

rule-21: if cycle 1 is complete and
        display(['The consultation is over.', nl])
        then the consultation is over.

/* ----- THE DEFAULT X ----- */

/* These default characteristics come into play when CWINE is unable
   to find either the best CHARACTERISTIC or the user's preferred
   CHARACTERISTIC. The system should never ask the user about these
   defaults, hence the 'noautomaticquestion' kb entry. */

noautomaticquestion(the default X).

the default body = medium.

the default color = red cf 50.
the default color = white cf 50.

the default sweetness = medium.

```

```

/* ----- THE FAULT WITH CYCLE N ----- */

multivalued(the fault with cycle N).

question(the fault with cycle N) =
    'Which characteristic of the wine would you like to change?'.

legalvals(the fault with cycle N) =
    [color, body, sweetness].

/* ----- THE MAIN COMPONENT ----- */

multivalued(the main component).

question(the main component) =
    'Is the main component of the meal meat, fish or poultry?'.

legalvals(the main component) =
    [meat, fish, poultry].

/* ----- THE NEW VALUE FOR X FOR CYCLE N ----- */

/*      The following question specification will work for color, body
and sweetness. Legalvals are still of course different, so those
specifications can't be collapsed.  */

question(the new value for X for cycle N) =    ['What ', X, ' would you
prefer?'].

legalvals(the new value for color for cycle N) =
    [red, white].

legalvals(the new value for body for cycle N) =
    [light, medium, full].

legalvals(the new value for sweetness for cycle N) =
    [dry, medium, sweet].

/* ----- THE PREFERRED BODY ----- */

multivalued(the preferred body).

question(the preferred body) =
    'Do you generally prefer light, medium or full bodied wines?'.

legalvals(the preferred body) =
    [light, medium, full].

```

/* ----- THE PREFERRED COLOR ----- */

multivalued(the preferred color).

question(the preferred color) =
 'Do you generally prefer red or white wines?'

legalvals(the preferred color) =
 [red, white].

/* ----- THE PREFERRED SWEETNESS ----- */

multivalued(the preferred sweetness).

question(the preferred sweetness) =
 'Do you generally prefer dry, medium or sweet wines?'

legalvals(the preferred sweetness) =
 [dry, medium, sweet].

/* ----- THE RECOMMENDED X FOR CYCLE N ----- */

/* These rules establish the recommended characteristics for the wines
that CWINE recommends during the first cycle (characteristics of
subsequent cycles are either carried over or explicitly stated by
the user).

 If the best characteristic is known, then that's recommended.
 If the best characteristic is NOT known, then the user's preference
 is recommended (provided the user states one). If neither the best
 nor the preferred characteristic is known, CWINE resorts to the
 default characteristics specified above. */

rule-22: if the best X = V
 then the recommended X for cycle 1 = V.

rule-23: if the best X is unknown and
 the preferred X = V
 then the recommended X for cycle 1 = V.

rule-24: if the best X is unknown and
 the preferred X is unknown and
 the default X = Y
 then the recommended X for cycle 1 = Y.

```

/* This rule carries over a characteristic from one cycle to the
   next, provided that the user doesn't want to change it. */

rule-25: if previouscycle to M = N and
         not(the fault with cycle N = X) and
         the recommended X for cycle N = V
         then the recommended X for cycle M = V.

/* This rule comes into play when the user indicates that a particular
   characteristic is objectionable. The user is asked what the new
   value for the characteristic should be, and that value is used in
   the next cycle of recommendations. */

rule-26: if previouscycle to M = N and
         the fault with cycle N = X and
         the new value for X for cycle M = V
         then the recommended X for cycle M = V.

/* ----- THE WINE FOR CYCLE N ----- */

multivalued(the wine for cycle N).

/* This rule uses recommends wines for a particular cycle by finding
   the characteristics of the wines to recommend and doing a 'lookup'
   operation in a small table of wines and recommends wines that match
   the characteristics provided. */

rule-27: if the recommended color for cycle N = C and
         the recommended body for cycle N = B and
         the recommended sweetness for cycle N = S and
         wine(C,B,S) = W
         then the wine for cycle N = W.

/* The following rule mentions 'feature', and hence is inconvenient
   to replace by a table entry. */

rule-28: if the recommended color for cycle N = white and
         the recommended body for cycle N = full and
         feature = spiciness
         then the wine for cycle N = gewuerztraminer.

/* ----- USER HAPPY WITH SELECTION N ----- */

question(user happy with selection N) = [nl,'Are you happy with these?'].

legalvals(user happy with selection N) = [yes,no].

```

/* ----- USER INFORMED OF SELECTION N ----- */

/* This rule uses 'mostlikely' to find the values of the characteristics of the wines that are believed with the most certainty. The values of these characteristics are output to the screen by means of a 'display' clause in the rule's premise. Note the use of 'do' to actually issue the 'show' command (ordinarily used only at the M.1 top level interpreter) from within the premise of the rule. */

```
rule-29: if mostlikely(the recommended color for cycle N) = C and
      mostlikely(the recommended body for cycle N) = B and
      mostlikely(the recommended sweetness for cycle N) = S and
      display(['The following wines will mostly be ',
               S, ', ', B, '-bodied, and ', C, '.'], nl,
               'They are recommended for your meal.', nl, nl]) and
      do(show the wine for cycle N)
      then user informed of selection N.
```

/* ----- WINE(X, Y, Z) ----- */

/* The following facts comprise a small tables that describe wines in terms of their characteristics. These facts are matched by rule-27 in order to come up with particular recommendations.

The 'noautomaticquestion' specification prevents M.1 from generating a question when the table doesn't contain a wine for a particular combination of characteristics. */

multivalued(wine(COLOR, BODY, SWEETNESS)).

noautomaticquestion(wine(COLOR, BODY, SWEETNESS)).

wine(red,medium,medium) = gamay.
wine(red,medium,sweet) = gamay.
wine(white,light,dry) = chablis.
wine(white,medium,dry) = 'sauvignon blanc'.
wine(white,medium,dry) = chardonnay.
wine(white,medium,medium) = chardonnay.
wine(white,full,dry) = chardonnay.
wine(white,full,medium) = chardonnay.
wine(white,light,dry) = soave.
wine(white,light,medium) = soave.
wine(white,light,medium) = riesling.
wine(white,light,sweet) = riesling.
wine(white,medium,medium) = riesling.
wine(white,medium,sweet) = riesling.
wine(white,light,medium) = 'chenin blanc'.
wine(white,light,sweet) = 'chenin blanc'.
wine(red,light,ANY) = valpolicella.
wine(red,ANY,dry) = 'cabernet sauvignon'.
wine(red,ANY,dry) = zinfandel.
wine(red,ANY,medium) = 'cabernet sauvignon'.
wine(red,ANY,medium) = zinfandel.
wine(red,medium,medium) = 'pinot noir'.
wine(red,full,ANY) = burgundy.

/* ----- */

Appendix I

Clocksin and Mellish Prolog Predicates

Below is a list of the 58 standard predicates listed in the book "Programming in Prolog" by William F Clocksin and C.S. Mellish. All predicates used in ABC are either part of the list below or are ABC defined predicates made from the predicates listed below.

!	(the cut)	integer
*	(multiplication)	is
+	(addition)	listing
-	(subtraction)	mod
.		name
/		nl
<		nonvar
=..		nospy
=<		not
==		notrace
=		op
>		put
>=		read
\==		reconsult
\=		repeat
append		retract
arg		see
asserta		seeing
assertz		seen
atom		skip
atomic		spy
call		tab
clause		tell
consult		telling
display		told
fail		trace
functor		true
get0		var
get		write

Vita

Eddy Gene Clark [REDACTED] He moved to Alabama and later to Tennessee as a young boy. He graduated from high school in Rutherford, Tennessee in 1973 and went to work as a cabinet builder. In the fall of 1974, he enlisted in the Air Force in the Traffic Management Office career field. He remained in the Air Force until the fall of 1980 when he left to enter College at the University of Tennessee at Martin to study pre-engineering. After two years at Martin, he went on to get his bachelor's degree in Electrical Engineering at the University of Tennessee at Knoxville where he graduated with honors. Upon graduation he attended Officer Training School and received his commission. His first assignment was working as a C-130/C-141 electrical engineer at the Air Logistic Center at Robins Air Force Base in Georgia. He was then accepted into the computer engineering master's program at the Air Force Institute of Technology.

[REDACTED]

[REDACTED]

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED			1b. RESTRICTIVE MARKINGS		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; distribution unlimited.		
2b. DECLASSIFICATION / DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S)		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCE/ENG/88D-2			7a. NAME OF MONITORING ORGANIZATION		
6a. NAME OF PERFORMING ORGANIZATION School of Engineering	6b. OFFICE SYMBOL (If applicable) AFIT/ENG		7b. ADDRESS (City, State, and ZIP Code) <i>Approved for release in accordance with AFR 190-1 S&P recommended 12 Jan 1989</i>		
6c. ADDRESS (City, State, and ZIP Code) Air Force Institute of Technology Wright Patterson AFB, OH 45433-6583		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER			
8a. NAME OF FUNDING / SPONSORING ORGANIZATION	8b. OFFICE SYMBOL (If applicable)		10. SOURCE OF FUNDING NUMBERS		
8c. ADDRESS (City, State, and ZIP Code)		PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification) An Educational Expert System Shell Integrating Object-Attribute-Value Triples and Frames					
12. PERSONAL AUTHOR(S) Eddy G. Clark, B.S., Capt, USAF					
13a. TYPE OF REPORT MSCE Thesis	13b. TIME COVERED FROM _____ TO _____		14. DATE OF REPORT (Year, Month, Day) 1988 December		15. PAGE COUNT
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)		
FIELD	GROUP	SUB-GROUP			
12	05		Artificial Intelligence, Expert Systems, Prolog		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) Thesis Chairman: Professor F.M. Brown Investigates the creation of an expert system shell which integrates object-attribute-value (OAV) triples with frames and implements the shell in standard Prolog. Additionally, the implemented expert system shell uses certainty factors, which allow it to perform inexact reasoning. The shell, named AFIT Backward Chainer, or ABC, represents its knowledge in facts, rules, and frames. ABC has an explanation facility that can explain how it derives a solution or why it asks particular questions when seeking information from the user. ABC expands upon an educational expert system shell called BC3. BC3, a rule-based shell developed at AFIT, symbolizes its knowledge with OAV triples. The development of ABC was accomplished using a rapid-prototype methodology.					
20. DISTRIBUTION / AVAILABILITY OF ABSTRACT <input checked="" type="checkbox"/> UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Eddy G. Clark, Captain, USAF			22b. TELEPHONE (Include Area Code) (513) 255-3030		22c. OFFICE SYMBOL AFIT/ENG